

# 解密搜索引擎 技术实战

## Lucene & Java 精华版

第2版

版本升级至  
Lucene 4

罗刚 等编著



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
http://www.phei.com.cn

# **解密搜索引擎技术实战： Lucene & Java精华版（第2版）**

罗刚 等编著

電子工業出版社

Publishing House of Electronics Industry  
北京 • BEIJING

## 内 容 简 介

本书是猎兔搜索开发团队的软件研发和教学实践的经验汇总。本书总结搜索引擎相关理论与实际解决方案,并给出了 Java 实现,其中利用了流行的开源项目 Lucene 和 Solr,而且还包括原创的实现。

本书主要包括总体介绍部分、爬虫部分、自然语言处理部分、全文检索部分以及相关案例分析。爬虫部分介绍了网页遍历方法和如何实现增量抓取,并介绍了从网页等各种格式的文档中提取主要内容的方法。自然语言处理部分从统计机器学习的原理出发,包括了中文分词与词性标注的理论与实现及在搜索引擎中的应用等细节,同时对文档排重、文本分类、自动聚类、句法分析树、拼写检查等自然语言处理领域的经典问题进行了深入浅出的介绍,并总结了实现方法。在全文检索部分,结合 Lucene 介绍了搜索引擎的原理与进展。用简单的例子介绍了 Lucene 的最新应用方法,包括完整的搜索实现过程:从完成索引到搜索用户界面的实现。此外还进一步介绍了实现准实时搜索的方法,展示了 Solr 的用法以及实现分布式搜索服务集群的方法。最后介绍了在地理信息系统领域和户外活动搜索领域的应用。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有,侵权必究。

## 图书在版编目(CIP)数据

解密搜索引擎技术实战: Lucene & Java 精华版 / 罗刚等编著. —2 版. —北京: 电子工业出版社, 2014.1  
ISBN 978-7-121-21732-6

I. ①解… II. ①罗… III. ①互联网络—情报检索 IV. ①G354.4

中国版本图书馆 CIP 数据核字(2013)第 251471 号

责任编辑: 董 英

印 刷: 北京中新伟业印刷有限公司

装 订: 北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×1092 1/16 印张: 32 字数: 678 千字

印 次: 2014 年 1 月第 1 次印刷

定 价: 79.00 元(含 DVD 光盘 1 张)

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。



# 前 言

**很多** 搜索相关的技术已经得到了初步的解决。在国内产业界也已经有很多公司掌握了基本的搜索开发技术并拥有专业的搜索技术开发人员。但是越来越多有价值的资讯对现有技术的处理能力仍然是一个挑战。

**为了** 方便实践，需要有良好实现的代码作为参考。为了节约篇幅，书中的代码只是核心片段。本书相关代码的完整版本在附带光盘中可以找到。

**作者** 罗刚在参加编写本书之前，还独立撰写过《自己动手写搜索引擎》一书，与王振东共同编写过《自己动手写网络爬虫》一书。经过 10 多年的技术积累以及猎兔搜索技术团队每年若干的研发投入，相信猎兔已经能够比以前做得更好。但越是深入接触客户的需求，越感觉到技术本身仍需要更多进步，才能满足实用的需要。写这本书也是考虑到，也许还需要更多的前进，才能使技术产生质的飞跃。

**本书** 分为相关技术总体介绍部分、爬虫部分、全文检索部分、自然语言处理部分以及相关案例分析部分。

**爬虫** 部分从基本的爬虫原理开始讲解，通过介绍优先级队列、宽度优先搜索等内容引领读者入门；之后根据当前风起云涌的云计算热潮，重点讲述了云计算的基本原理及其在搜索中的应用，以及 Web 图分析、信息抽取等内容；为了能够让读者更深入地了解爬虫，本书还介绍了有关爬虫的数据挖掘的内容。

**全文** 检索部分重点介绍了搜索的基本原理与使用。主要介绍了开源软件实现 Lucene 以及 Solr。不仅介绍了如何使用这些开源软件，而且还介绍了其中的一些实现原理。Lucene 更高版本的改进指出了当前需要解决的问题，欢迎读者在了解基本原理后进行更深入的研究。

**自然**语言处理部分向来是笔者关注的重点，因为系统的智能化依赖于此。开发中文搜索离不开中文分词。开发任何自然语言的搜索也离不开对相应语言的处理。对自然语言的处理其实也可以用到对 Java 或 C 语言这样的机器语言的处理方法，只不过处理自然语言更难一点。

**虽然**本书的每个章节都已经用代码强化了实现细节，但是对于初学者来说，也许需要更多的案例来理解相关技术在真实场景中的用法。案例分析部分介绍了在地理信息系统领域和户外活动搜索领域的应用。

**本书**适合需要具体实现搜索引擎的程序员使用，对于信息检索等相关研究人员也有一定的参考价值，同时猎兔搜索技术团队也已经开发出以本书为基础的专门培训课程和商业软件。

**高级**开发人员也可以参加猎兔的培训或者创业团队。职场人员经常面临各种压力。选择猎兔培训，不是几个月学完以后就不再见，而是给大家提供持久的支持。当以后需要再次找工作的时候，或者需要创业时，依然可以在这里找到支持。很多商业运营的大项目失败的代价太高，所以他们往往只招有多年开发经验的工程师。但是为了成长就不要怕犯错误，在培训时可以等学员犯了错误之后再告知正确答案。有经验的工程师也可以在这里学习到完整的技术体系。

**感谢**开源软件开发人员和家人、关心猎兔的老师和朋友、创业伙伴以及信赖猎兔软件的客户多年来的支持。可以通过 QQ(270954928)联系作者，或者加 QQ 群(166015123)讨论相关技术。参与本书编写的有罗刚、石天盈、张继红、赵艺、张江波、赵英晨、高丹丹、徐友峰、孙宽、王提榼，在此一并表示感谢。

编 者



# 目 录

第 1 章 搜索引擎总体结构 .....	1	2.4.2 布隆过滤器 .....	42
1.1 搜索引擎基本模块 .....	1	2.5 并行抓取 .....	45
1.2 开发环境 .....	2	2.5.1 多线程爬虫 .....	46
1.3 搜索引擎工作原理 .....	3	2.5.2 垂直搜索的多线程爬虫 .....	48
1.3.1 网络爬虫 .....	4	2.5.3 异步 I/O .....	49
1.3.2 全文索引结构与 Lucene 实现 .....	4	2.6 RSS 抓取 .....	53
1.3.3 搜索用户界面 .....	7	2.7 抓取 FTP .....	55
1.3.4 计算框架 .....	8	2.8 下载图片 .....	55
1.3.5 文本挖掘 .....	9	2.9 图像的 OCR 识别 .....	56
1.4 本章小结 .....	9	2.9.1 图像二值化 .....	57
第 2 章 网络爬虫的原理与应用 .....	11	2.9.2 切分图像 .....	60
2.1 爬虫的基本原理 .....	11	2.9.3 SVM 分类 .....	63
2.2 爬虫架构 .....	14	2.10 Web 结构挖掘 .....	67
2.2.1 基本架构 .....	14	2.10.1 存储 Web 图 .....	67
2.2.2 分布式爬虫架构 .....	16	2.10.2 PageRank 算法 .....	71
2.2.3 垂直爬虫架构 .....	17	2.10.3 HITS 算法 .....	77
2.3 抓取网页 .....	18	2.10.4 主题相关的 PageRank .....	81
2.3.1 下载网页的基本方法 .....	19	2.11 部署爬虫 .....	83
2.3.2 网页更新 .....	23	2.12 本章小结 .....	83
2.3.3 抓取限制应对方法 .....	25	第 3 章 索引内容提取 .....	86
2.3.4 URL 地址提取 .....	28	3.1 从 HTML 文件中提取文本 .....	86
2.3.5 抓取 JavaScript 动态页面 .....	28	3.1.1 识别网页的编码 .....	86
2.3.6 抓取即时信息 .....	31	3.1.2 网页编码转换为字符串编码 .....	89
2.3.7 抓取暗网 .....	32	3.1.3 使用正则表达式提取数据 .....	89
2.3.8 信息过滤 .....	33	3.1.4 结构化信息提取 .....	91
2.3.9 最好优先遍历 .....	39	3.1.5 网页的 DOM 结构 .....	94
2.4 存储 URL 地址 .....	40	3.1.6 使用 NekoHTML 提取信息 .....	95
2.4.1 BerkeleyDB .....	40	3.1.7 使用 Jsoup 提取信息 .....	101

3.1.8	网页去噪	105	4.11	平滑算法	194
3.1.9	网页结构相似度计算	110	4.12	本章小结	198
3.1.10	提取标题	112	第 5 章	让搜索引擎理解自然语言	199
3.1.11	提取日期	113	5.1	停用词表	200
3.2	从非 HTML 文件中提取文本	113	5.2	句法分析树	201
3.2.1	提取标题的一般方法	114	5.3	相似度计算	205
3.2.2	PDF 文件	118	5.4	文档排重	209
3.2.3	Word 文件	122	5.4.1	语义指纹	210
3.2.4	Rtf 文件	123	5.4.2	SimHash	213
3.2.5	Excel 文件	134	5.4.3	分布式文档排重	223
3.2.6	PowerPoint 文件	137	5.5	中文关键词提取	223
3.3	流媒体内容提取	137	5.5.1	关键词提取的基本方法	223
3.3.1	音频流内容提取	138	5.5.2	HITS 算法应用于 关键词提取	226
3.3.2	视频流内容提取	140	5.5.3	从网页中提取关键词	228
3.4	存储提取内容	142	5.6	相关搜索词	228
3.5	本章小结	143	5.6.1	挖掘相关搜索词	229
第 4 章	中文分词的原理与实现	144	5.6.2	使用多线程计算 相关搜索词	231
4.1	Lucene 中的中文分词	145	5.7	信息提取	232
4.1.1	Lucene 切分原理	145	5.8	拼写检查与建议	237
4.1.2	Lucene 中的 Analyzer	146	5.8.1	模糊匹配问题	240
4.1.3	自己写 Analyzer	148	5.8.2	英文拼写检查	242
4.1.4	Lietu 中文分词	150	5.8.3	中文拼写检查	244
4.2	查找词典算法	151	5.9	自动摘要	247
4.2.1	标准 Trie 树	151	5.9.1	自动摘要技术	247
4.2.2	三叉 Trie 树	154	5.9.2	自动摘要的设计	247
4.3	中文分词的原理	159	5.9.3	Lucene 中的动态摘要	254
4.4	中文分词流程与结构	162	5.10	文本分类	257
4.5	形成切分词图	164	5.10.1	特征提取	259
4.6	概率语言模型的分词方法	170	5.10.2	中心向量法	262
4.7	$N$ 元分词方法	174	5.10.3	朴素贝叶斯	265
4.8	新词发现	178	5.10.4	支持向量机	272
4.9	未登录词识别	180	5.10.5	规则方法	279
4.10	词性标注	181	5.10.6	网页分类	282
4.10.1	隐马尔可夫模型	184	5.11	拼音转换	283
4.10.2	基于转换的错误 学习方法	192			

5.12	概念搜索	284	6.4.8	FieldScoreQuery	353
5.13	多语言搜索	292	6.5	读写并发控制	356
5.14	跨语言搜索	293	6.6	检索模型	356
5.15	情感识别	295	6.6.1	向量空间模型	357
5.15.1	确定词语的褒贬倾向	298	6.6.2	BM25 概率模型	361
5.15.2	实现情感识别	300	6.6.3	统计语言模型	367
5.16	本章小结	301	6.7	本章小结	369
第 6 章	Lucene 原理与应用	303	第 7 章	搜索引擎用户界面	370
6.1	Lucene 深入介绍	304	7.1	实现 Lucene 搜索	370
6.1.1	常用查询对象	304	7.2	实现搜索接口	372
6.1.2	查询语法与解析	304	7.2.1	编码识别	372
6.1.3	查询原理	308	7.2.2	布尔搜索	375
6.1.4	分析文本	309	7.2.3	指定范围搜索	375
6.1.5	使用 Filter 筛选搜索结果	316	7.2.4	搜索结果排序	376
6.1.6	遍历索引库	317	7.2.5	搜索页面的索引缓存与更新	377
6.1.7	索引数值列	318	7.3	历史搜索词记录	380
6.2	Lucene 中的压缩算法	322	7.4	实现关键词高亮显示	381
6.2.1	变长压缩	322	7.5	实现分类统计视图	383
6.2.2	PForDelta	324	7.6	实现 Ajax 搜索联想词	388
6.2.3	前缀压缩	326	7.6.1	估计查询词的文档频率	388
6.2.4	差分编码	328	7.6.2	搜索联想词总体结构	389
6.3	创建和维护索引库	330	7.6.3	服务器端处理	389
6.3.1	创建索引库	330	7.6.4	浏览器端处理	390
6.3.2	向索引库中添加索引文档	331	7.6.5	服务器端改进	395
6.3.3	删除索引库中的索引文档	334	7.6.6	拼音提示	398
6.3.4	更新索引库中的索引文档	334	7.6.7	部署总结	399
6.3.5	索引的合并	335	7.7	集成其他功能	399
6.3.6	索引文件格式	335	7.7.1	拼写检查	399
6.4	查找索引库	338	7.7.2	分类统计	400
6.4.1	查询过程	338	7.7.3	相关搜索	402
6.4.2	常用查询	342	7.7.4	再次查找	405
6.4.3	基本词查询	343	7.7.5	搜索日志	405
6.4.4	模糊匹配	343	7.8	搜索日志分析	407
6.4.5	布尔查询	345	7.8.1	日志信息过滤	407
6.4.6	短语查询	347	7.8.2	信息统计	409
6.4.7	跨度查询	349			



7.8.3 挖掘日志信息·····	411	8.3.8 扩展 Solr·····	467
7.9 本章小结·····	412	8.3.9 查询 Web 图·····	471
<b>第 8 章 使用 Solr 实现企业搜索·····</b>	<b>413</b>	8.4 本章小结·····	473
8.1 Solr 简介·····	413	<b>第 9 章 地理信息系统案例分析·····</b>	<b>474</b>
8.2 Solr 基本用法·····	414	9.1 新闻提取·····	474
8.2.1 Solr 服务器端的配置与 中文支持·····	415	9.2 POI 信息提取·····	479
8.2.2 把数据放进 Solr·····	421	9.2.1 提取主体·····	484
8.2.3 删除数据·····	423	9.2.2 提取地区·····	485
8.2.4 Solr 客户端与搜索界面·····	424	9.2.3 指代消解·····	487
8.2.5 Spring 实现的搜索界面·····	425	9.3 机器翻译·····	489
8.2.6 Solr 索引库的查找·····	436	9.3.1 词对齐·····	490
8.2.7 索引分发·····	440	9.3.2 翻译公司名·····	491
8.2.8 Solr 搜索优化·····	442	9.3.3 调整语序·····	493
8.3 Solr 扩展与定制·····	445	9.4 本章小结·····	494
8.3.1 Solr 中字词混合索引·····	445	<b>第 10 章 户外活动搜索案例分析·····</b>	<b>495</b>
8.3.2 相关检索·····	447	10.1 爬虫·····	495
8.3.3 搜索结果去重·····	449	10.2 信息提取·····	497
8.3.4 定制输入输出·····	453	10.3 活动分类·····	500
8.3.5 分布式搜索·····	457	10.4 搜索·····	501
8.3.6 SolrJ 查询分析器·····	458	10.5 本章小结·····	501
8.3.7 扩展 SolrJ·····	466	<b>参考资料·····</b>	<b>502</b>



## 第 1 章

# 搜索引擎总体结构

本章首先概要地介绍搜索引擎的总体结构和基本模块，然后介绍其中最核心的模块全文检索的基本原理。为了尽快普及搜索引擎开发技术，本章介绍的搜索引擎结构可以采用开源软件实现。为了通过实践来深入了解相关技术，本章中会介绍相关的开发环境。本书介绍的搜索技术使用 Java 编程语言实现，之所以没有采用性能可能会更好的 C/C++，是希望读者不仅能够快速完成相关的开发任务，而且可以把相关实践作为一个容易上手的游戏。另外，为了集中关注程序的基本逻辑，书中的 Java 代码去掉了一些错误和异常处理，实际可以运行的代码可以在本书附带的光盘中找到。在后面的各章中会深入探索搜索引擎的每个组成模块。



### 1.1 搜索引擎基本模块

一个最简单的搜索引擎由索引和搜索界面两部分组成，相对完整的搜索结构如图 1-1 所示。

实现按关键字快速搜索的方法是建立全文索引库，所以最基础的程序是管理全文索引库的程序。搜索的数据来源可以是互联网或者数据库，也可以是本地路径等。搜索引擎的基本模块结构如图 1-2 所示。

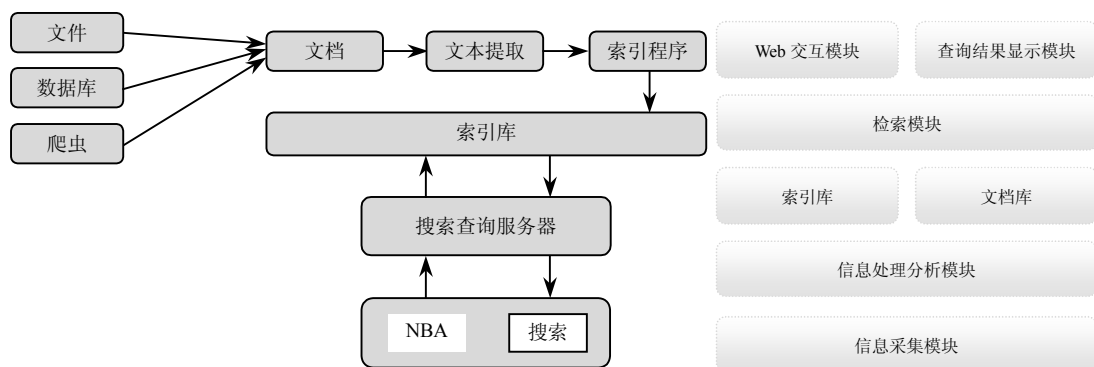


图 1-1 搜索引擎的简单结构

图 1-2 搜索引擎中的主要模块



## 1.2 开发环境

由于开源软件的迅速发展，可以借助开源软件简化搜索引擎开发工作。很多开源软件用 Java 语言开发，例如最流行的全文索引库 Lucene，所以本书采用 Java 来实现搜索。为了实现一个简单的指定目录文件的搜索引擎，首先要准备好 JDK 和集成开发环境 Eclipse。当前可以使用 JDK 1.7。JDK 1.7 可以从 Java 官方网站 <http://www.oracle.com/technetwork/java/index.html> 下载得到。使用默认方式安装即可。本书中的程序在附带光盘中都能找到，可以直接导入到 Eclipse 中。Eclipse 默认是英文界面，如果习惯用中文界面可以从 <http://www.eclipse.org/babel/downloads.php> 下载支持中文的语言包。

Lucene 是一个 Java 实现的 jar 包用来管理搜索引擎索引库。可以从 <http://lucene.apache.org/core/> 下载到最新版本的 Lucene，当前的版本是 4.3.1。

如果需要用 Web 界面搜索，还要下载 Tomcat，当前可以从 <http://tomcat.apache.org/> 下载到，推荐使用 Tomcat 6 以上的版本。使用开源的全文检索包 Lucene 做索引后，要把实现搜索的界面发布到 Tomcat。

Lucene 及一些相关项目的源代码由版本管理工具 SVN 管理，如果要构建源代码工程，可以使用工具 Ant 和 Maven。

如果需要导出 Lucene 的最新开发版本，就需要用到 SVN 的客户端。小乌龟 TortoiseSVN 是最流行的 SVN 客户端。TortoiseSVN 的下载地址是 <http://tortoisesvn.tigris.org/>。安装 TortoiseSVN 后，选择一个存放源代码的文件夹，单击鼠标右键，选择弹出菜单中的 Export

选项，导出源代码界面如图 1-3 所示。



图 1-3 TortoiseSVN 导出代码界面

Ant 与 Maven 都和项目管理软件 make 类似。虽然 Maven 正在逐步替代 Ant，但当前仍然有很多开源项目在继续使用 Ant。从 <http://ant.apache.org/bindownload.cgi> 可以下载到 Ant 的最新版本。

在 Windows 下 ant.bat 和三个环境变量相关，即 ANT\_HOME、CLASSPATH 和 JAVA\_HOME。需要用路径设置 ANT\_HOME 和 JAVA\_HOME 环境变量，并且路径不要以\或/结束，不要设置 CLASSPATH。如果把 Ant 解压到 C:\apache-ant-1.7.1，则修改环境变量 PATH，增加当前路径 C:\apache-ant-1.7.1\bin。大部分用 Ant 构建的项目只需要如下命令即可：

```
#ant
```

可以从 <http://maven.apache.org/download.html> 下载最新版本的 Maven，当前版本是 maven-2.2.1。解压下载的 Maven 压缩文件到 C 盘根路径，将创建一个 C:\apache-maven-2.2.1 路径。修改 Windows 系统环境变量 PATH，增加当前路径 C:\apache-maven-2.2.1\bin。大部分用 Maven 构建的项目只需要如下命令即可：

```
#mvn clean install
```



### 1.3 搜索引擎工作原理

一个基本的搜索包括采集数据的爬虫和索引库管理以及搜索页面展现等部分。

### 1.3.1 网络爬虫

网络爬虫（Crawler）又被称作网络机器人（Robot）或者蜘蛛（Spider），它的主要目的是为获取在互联网上的信息。只有掌握了“吸星大法”，才能源源不断地获取信息。网络爬虫利用网页中的超链接遍历互联网，通过 URL 引用从一个 HTML 文档爬行到另一个 HTML 文档。<http://dmoz.org> 可以作为整个互联网抓取的入口。网络爬虫收集到的信息可有多种用途，如建立索引、HTML 文件的验证、URL 链接验证、获取更新信息、站点镜像等。为了检查网页内容是否更新过，网络爬虫建立的页面数据库往往包含根据页面内容生成的文摘。

在抓取网页时大部分网络爬虫会遵循 Robot.txt 协议。网站本身可以有两种方式声明不想被搜索引擎收入的内容：第一种方式是在站点的根目录下增加一个纯文本文件 <http://www.yourdomain.com/robots.txt>；另外一种方式是直接在 HTML 页面中使用 robots 的 meta 标签。

### 1.3.2 全文索引结构与 Lucene 实现

为了方便查询，早在计算机出现之前就出现了人工为图书建立的索引，比如图 1-4 中的名词索引。

为了按词快速定位抓取过来的文档，需要以词为基础建立全文索引，也叫倒排索引（Inverted index），如图 1-5 所示。

——按名词的拼音顺序检索	
名词索引	
半地下室 semi-basement	
房间地面低于室外地平面的高度超过该房间净高的1/3，且不超过1/2者。 （摘自《住宅设计规范》（GB 50096-1999）3页 中国建筑工业出版社 1999.5第一版）	
壁柜 cabinet	
住宅套内与墙壁结合而成的落地贮藏空间。 （摘自《住宅设计规范》（GB 50096-1999）3页 中国建筑工业出版社 1999.5第一版）	
比例 proportion	
建筑构成各部分和各部分之间的相互关系，以及各部分与整体之间的比较关系。建筑比例是建筑构成中的一种量度尺度，有了具体的尺度才具有比例的真正意义。 （摘自《中国土木建筑百科全书》（建筑卷）24页 中国建筑工业出版社 1995.5第一版）	
变形缝 Deformation joint	
为防止建筑物在外界因素作用下，结构内部产生附加变形和应力，导致开裂甚至破坏而预留的构造缝。变形缝包括 （摘自《中国土木建筑百科全书》（建筑卷）24页 中国建筑工业出版社 1995.5第一版）	
标准层 typical floor	
平面布置相同的住宅楼层。 （摘自《住宅设计规范》（GB 50096-1999）2页 中国建筑工业出版社 1999.5第一版）	
不对称均衡 asymmetrical balance	

图 1-4 人工建立的名词索引

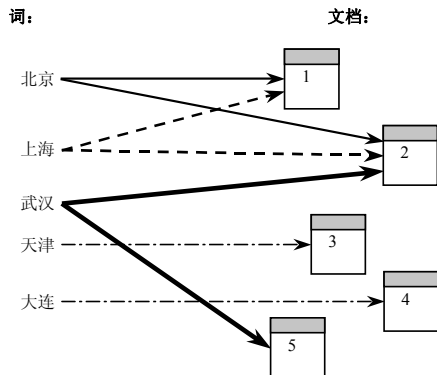


图 1-5 以词为基础的全文索引

倒排索引是相对于正向索引来说的，首先用正向索引来存储每个文档对应的单词列表，然后再建立倒排索引，根据单词来索引文档编号。

例如要索引如下两个文档：

Doc1：自己动手写搜索引擎  
Doc2：自己动手写网络爬虫

在 Lucene 中的倒排索引结构如表 1-1 所示。

每个单词（term）后面的文档编号（docId）列表叫做投递列表（posting list）。在 Lucene 中，倒排索引结构存储在二进制格式的多个索引文件中，其中以 tis 为后缀的文件中包含了单词信息；以 frq 为后缀的文件记录单词的文档编号和这个单词在文档中出现了多少次，也就是频率信息；以 prx 为后缀的文件包含了单词出现的位置信息。

表 1-1 Lucene 中的倒排索引结构

词	(文档, 频率)	位 置
动手	(1, 1),(2, 1)	(2),(2)
搜索引擎	(1, 1)	(4)
网络爬虫	(2, 1)	(4)
写	(1, 1),(2, 1)	(3),(3)
自己	(1, 1),(2, 1)	(1),(1)

为了快速查找单词，可以先对单词列表排序，然后再折半查找已经排好序的词表。下面是实现折半查找的代码。

```
int low = fromIndex; //开始位置
int high = toIndex - 1; //结束位置

while (low <= high) {
    int mid = (low + high) >> 1; //相当于 mid = (low + high)/2
    Comparable midVal = (Comparable)a[mid]; //取中间的值
    int cmp = midVal.compareTo(key); //中间值和要找的关键字比较

    if (cmp < 0)
        low = mid + 1;
    else if (cmp > 0)
        high = mid - 1;
    else
        return mid; //查找成功，返回找到的位置
}
return -(low + 1); //没找到，返回负值
```

词表是类似这样的数据结构：SortedMap<Term,Postings>。如果词表很小，内存能够放下，则可以使用折半查找算法来查询一个词对应的文档序列。如果内存不能完全放下倒排

索引中的词表，如何利用索引文件查找呢？理论上，可以采用 B 树存储词表。为了简化实现，Lucene3 以前的版本使用了跳表。跳表的实现把词组织成固定大小的块，例如每 32 个词放入一个新的块，然后在块上建立一个索引，记住每个块的开始词在文件中的位置。

Lucene (<http://lucene.apache.org/>) 是一个开放源代码的全文索引库。经过 10 多年的发展，Lucene 拥有了大量的用户和活跃的开发团队。Eclipse 软件和 Twitter 网站等都在使用 Lucene。如果说 Google 是拥有最多用户访问的搜索引擎网站，那么拥有最多开发人员支持的搜索软件项目也许是 Lucene。

Lucene 的整体结构如图 1-6 所示。

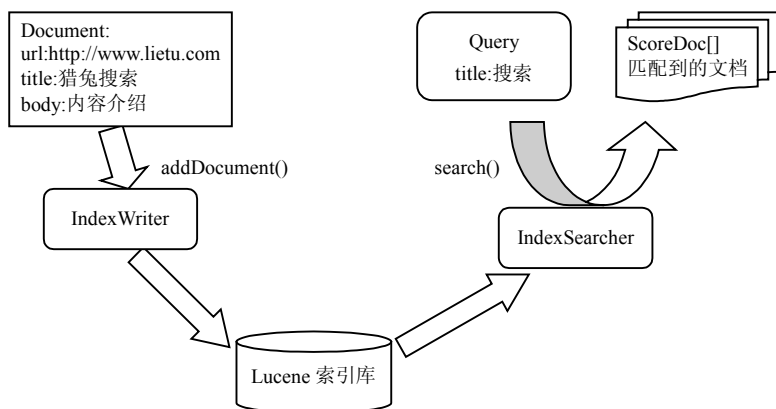


图 1-6 Lucene 原理图

Lucene 中的基本概念介绍如下。

- 第一个概念是 Index，也就是索引库。文档的集合组成索引。和一般的数据库不一样，Lucene 不支持定义主键。在 Lucene 中并不存在一个叫做 Index 的类。通过 IndexWriter 来写索引，通过 IndexReader 来读索引。索引库在物理形式上一般是位于一个路径下的一系列文件。
- 一段有意义的文字需要通过 Analyzer 分割成一个个词语后才能按关键词搜索。Analyzer 就是分析器，StandardAnalyzer 是 Lucene 中最常用的分析器。为了达到更好的搜索效果，不同的语言可以使用不同的分析器，例如 CnAnalyzer 就是一个主要处理中文的分析器。
- Analyzer 返回的结果就是一串 Token。Token 包含一个代表词本身含义的字符串和该词在文章中相应的起止偏移位置，Token 还包含一个用来存储词类型的字符串。
- 一个 Document 代表索引库中的一条记录，也叫做文档。要搜索的信息封装成

Document 后通过 IndexWriter 写入索引库。调用 Searcher 接口按关键词搜索后，返回的也是一个封装后的 Document 列表。

- 一个 Document 可以包含多个列，叫做 Field。例如一篇文章可以包含“标题”、“正文”、“修改时间”等 Field。创建这些列对象以后，可以通过 Document 的 add 方法增加这些列。和一般的数据库不一样，一个文档的一个列可以有多个值。例如一篇文档既可以属于互联网类，又可以属于科技类。
- Term 是搜索语法的最小单位，复杂的搜索语法会分解成一个 Term 查询。它表示文档的一个词语，Term 由两部分组成：它表示的词语和这个词语所出现的 Field。

Lucene 中的 API 相对数据库来说比较灵活，没有类似数据库先定义表结构后使用的过程。如果前后两次写索引时定义的列名称不一样，Lucene 会自动创建新的列，所以 Field 的一致性需要我们自己掌握。

### 1.3.3 搜索用户界面

随着搜索引擎技术逐渐走向成熟，搜索用户界面也形成了一些比较固定的模式。

- 输入提示词：用户在搜索框中输入查询词的过程中随时给予查询提示词。对中文来说，当用户输入拼音时，也能提示。
- 相关搜索提示词：当用户对当前搜索结果不满意时，也许换一个搜索词就能够得到更有用的信息。一般会根据用户当前搜索词给出多个相关的提示词。可以看成是协同过滤在搜索词上的一种具体应用。
- 相关文档：返回和搜索结果中的某一个文档相似的文档。例如：Google 搜索结果中的“类似结果”。
- 在结果中查询：如果返回结果很多，则用户在返回结果中再次输入查询词以缩小查询范围。
- 分类统计：返回搜索结果在类别中的分布图。用户可以按类别缩小搜索范围，或者在搜索结果中导航。有点类似数据仓库中的向下钻取和向上钻取。
- 搜索热词统计界面：往往按用户类别统计搜索词，例如按用户所属区域或者按用户所属部门等，当然也可以直接按用户统计搜索热词，例如 Google 的 Trends。

搜索界面的改进都是以用户体验为导向，所以搜索用户界面往往还会根据具体应用场景优化。所有这一切都是为了和用户的交互达到最优的效果。



### 1.3.4 计算框架

互联网搜索经常面临海量数据，需要分布式的计算框架来执行对网页重要度打分等计算。有的计算数据很少，但是计算量很大；还有些计算数据量比较大，但是计算量相对比较小。例如计算圆周率是计算密集型，互联网搜索中的计算往往是数据密集型，所以出现了数据密集型的云计算框架。MapReduce 是一种常用的云计算框架。

MapReduce 把计算任务分成两个阶段。映射（Map）阶段按数据分类完成基本计算；化简（Reduce）阶段收集基本的计算结果。使用 MapReduce 统计词频的例子如图 1-7 所示。

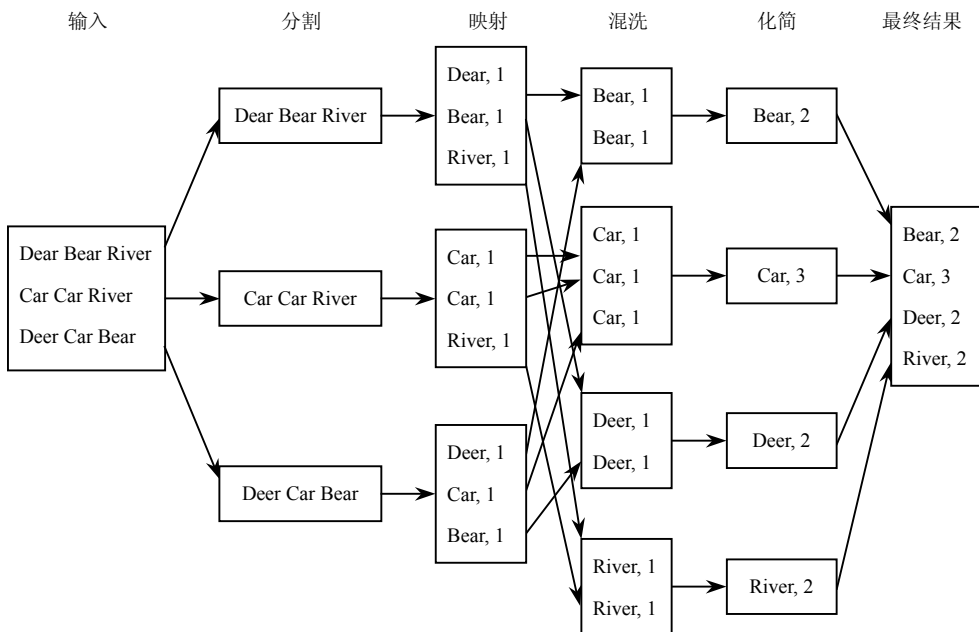


图 1-7 词频统计的例子

Hadoop (<http://hadoop.apache.org/>) 是 MapReduce 思想实现的一个开源计算平台，已经在包括百度等搜索引擎开发公司得到商用。但是 MapReduce 是批处理的操作方式，一般来说，直到完成上一阶段的操作后才能启动下一阶段的操作。

我们需要有一种计算，可以尽快出结果，随着时间的延长，计算结果会越来越好。很多计算可以用迭代的方式做，迭代次数越多，结果往往越好，比如 PageRank 或者 KMeans、EM 算法。当然，这个应该不只需要迭代，还需要向最优解收敛。

1.3.5 文本挖掘

搜索文本信息需要理解人类的自然语言。文本挖掘指从大量文本数据中抽取隐含的、未知的、可能有用的信息。

常用的文本挖掘方法包括：全文检索、中文分词、句法分析、文本分类、文本聚类、关键词提取、文本摘要、信息提取、智能问答等。文本挖掘相关技术的结构如图 1-8 所示。

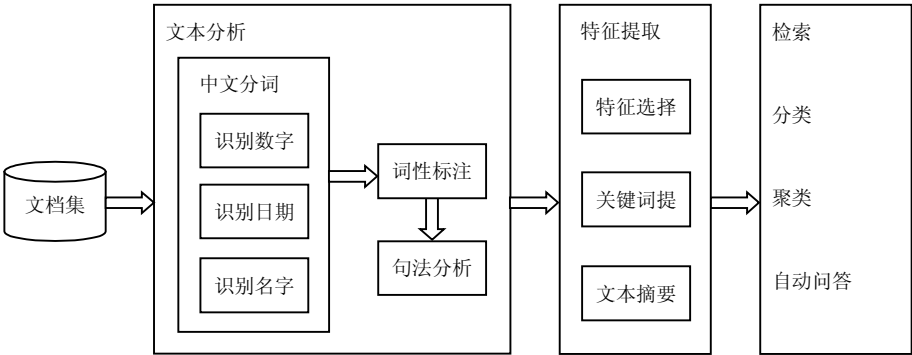


图 1-8 文本挖掘的结构



1.4 本章小结

本章介绍了互联网搜索 Google 及其创新原则。在 Google 出现之前，Yahoo 使用人工对网站进行分类，提供按目录导航和搜索目录数据库等功能。在 Google 尚未占据互联网搜索绝对优势之前，也是在笔者第一次听人推荐 Google 之前，就出现了元搜索引擎（Meta Search Engine）。用户只需提交一次搜索请求，由元搜索引擎负责转换处理后提交给多个预先选定的独立搜索引擎，并将从各独立搜索引擎返回的所有查询结果集中起来处理后再返回给用户。但 Google 开始独家垄断全球互联网搜索后，元搜索引擎逐渐被人遗忘。

Google 早期使用 MapReduce 实现分布式索引。后来之所以放弃这种方式，是因为它并不能为 Google 提供它想要的索引速度。工程师需要等待 8 个小时的计算时间才能够得到计算的全部结果，然后把它发布到索引系统中去。随着实时检索时代的到来，Google 需要在几秒内刷新索引内容，而非 8 小时。



Hadoop 来源于开源的分布式搜索项目 Nutch。Powerset 公司在 Hadoop 的基础上开发了基于 BigTable 架构的数据库 Hbase (<http://hbase.apache.org/>)。2008 年，微软收购了 Powerset。

与文本挖掘技术对应的是包括语音识别、基于内容的图像检索等技术的流媒体挖掘技术。随着网络电视和视频网站的流行，流媒体挖掘技术正越来越引起人们的关注。

除了像 Google 的网页搜索这样的常规搜索引擎，还有些特殊的搜索引擎。搜索的输入不一定是简单的关键词，例如 Wolfram|Alpha (<http://www.wolframalpha.com/>) 是一个特殊的可计算的知识引擎，它可以根据用户问句式的输入精确地返回一个答案。TextRunner 搜索 (<http://www.cs.washington.edu/research/textrunner/>) 是另外一个问答式的搜索。搜索引擎不一定只是简单列出搜索结果，Vivisimo (<http://vivisimo.com/>) 是实现聚类的搜索，也可以分类统计搜索结果。Vivisimo 后来被 IBM 收购以推动大数据发展。搜索结果不一定是文档，例如 Aardvark (<http://vark.com/>) 是一个社会化搜索，它可以自动选择合适的人来回答用户提出的问题。



## 第 2 章

# 网络爬虫的原理与应用

网络爬虫从互联网中的海量信息源源不断地抓取有用信息，搜索引擎结果中的信息都来源于此。如果把互联网比喻成一个覆盖地球的蜘蛛网，那么抓取程序就是在网上爬来爬去的蜘蛛。爬虫的好处是能汇集有用的数据。

网络爬虫需要实现的基本功能包括下载网页以及对 URL 地址的遍历。为了高效快速地遍历大量的 URL 地址，还需要应用专门的数据结构来优化。又因为爬虫很消耗带宽资源，所以设计爬虫时需要仔细考虑如何节省网络带宽。



### 2.1 爬虫的基本原理

如果把网页看成节点，网页之间的超链接看成边，则可以把整个互联网看成是一个巨大的非连通图。为了获取网页，需要有一个初始的 URL 地址列表。然后通过网页中的超链接访问到其他页面。有人可能会奇怪像 Google 或百度这样的搜索门户怎么设置这个初始的 URL 地址列表。一般来说，网站所有者把网站提交给分类目录，例如 dmoz (<http://www.dmoz.org/>)，爬虫则可以从开放式分类目录 dmoz 抓取。

抓取下来的网页中包含了想要的信息，一般存放在数据库或索引库这样专门的存储系统中，如图 2-1 所示。

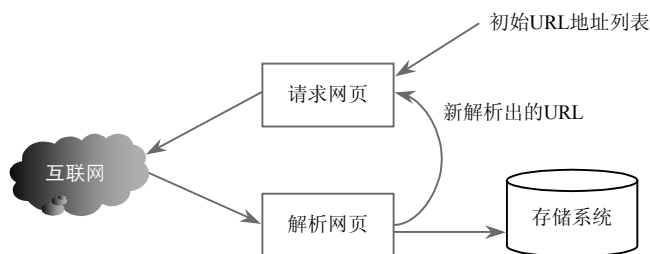


图 2-1 爬虫基本结构图

在搜索引擎中，爬虫程序是从一系列种子链接把这些初始网页中的 URL 提取出来，放入 URL 工作队列（Todo 队列，又叫做 Frontier），然后遍历所有工作队列中的 URL，下载网页并把其中新发现的 URL 再次放入工作队列。为了判断一个 URL 是否已经遍历过，应把所有遍历过的 URL 放入历史表（Visited 表）。爬虫抓取的基本过程如图 2-2 所示。

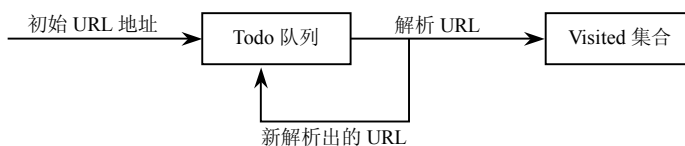


图 2-2 网页遍历流程图

抓取的主要流程如下：

```
while (todo.size() > 0) { //如果 Todo 队列不是空的
    //从 Todo 队列里面提取 URL
    String strUrl = todo.iterator().next();
    //下载 URL 对应的网页内容
    String content = downloadPageContent(strUrl);

    //提取网页内容中新发现的 URL 链接
    HashSet<String> newLinks = retrieveLinks(content, new URL(strUrl));
    //把新发现的链接加入 Todo 队列
    todo.addAll(newLinks);

    //从 Todo 队列里删除已经爬过的 URL
    todo.remove(strUrl);
    //把从 Todo 队列里删除的 URL 添加到 Visited 集合中
    visited.add(strUrl);
}
```

整个互联网是一个大的图，其中，每个 URL 相当于图中的一个节点，因此，网页遍历就可以采用图遍历的算法进行。通常，网络爬虫的遍历策略有三种：广度优先遍历、深度

优先遍历和最佳优先遍历。其中，深度优先遍历由于极有可能使爬虫陷入黑洞，因此，广度优先遍历和最佳优先遍历就成为常用的爬虫策略。

广度优先遍历是指网络爬虫会先抓取起始网页中链接的所有网页，然后再选择其中的一个链接网页，继续抓取在此网页中链接的所有网页。这是最常用的方式，这个方法也可以让网络爬虫并行处理，提高其抓取速度。图 2-3 说明广度遍历的过程。

例如在图 2-3 中，A 为种子节点，则首先遍历 A（第一层），接着是 BCDEF（第二层），然后遍历 GH（第三层），最后遍历 I（第四层）。

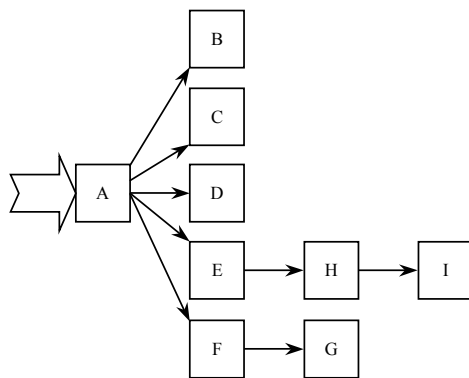


图 2-3 网络爬虫遍历的图

广度优先遍历使用一个队列来实现 Todo 表，先访问的网页先扩展。针对图 2-3，广度优先遍历的执行过程如表 2-1 所示。

表 2-1 广度优先遍历过程表

Todo 队列	Visited 集合	Todo 队列	Visited 集合
A	null	F H	A B C D E
B C D E F	A	H G	A B C D E F
C D E F	A B	G I	A B C D E F H
D E F	A B C	I	A B C D E F H G
E F	A B C D	null	A B C D E F H G I

庄子曾说：“吾生也有涯，而知也无涯，以有涯随无涯，殆已。”在学习和工作的时候，需要分辨事情的轻重缓急，否则一味蛮干，最终结果只能是“殆已”。对于浩瀚无边的互联网而言，网络爬虫涉及的页面只是冰山一角。因此，需要以最小的代价（硬件、带宽）获取到最大的利益（数量最多的重要网页）。

为了先抓取重要的网页，可以采用最佳优先爬虫策略。最佳优先爬虫策略也称为“页面选择问题”（Page Selection），通常保证在有限带宽条件下，尽可能地照顾到重要性高的网页。

如何实现最佳优先爬虫呢，最简单的方式可以使用优先级队列（Priority Queue）来实现 Todo 表，这样，每次选出来扩展的 URL 就是具有最高重要性的网页。在队列中，先进入的元素先出，但是在优先队列中，优先级高的元素先出队列。



比如，假设图 2-3 的节点重要性为  $D > B > C > A > E > F > I > G > H$ ，则整个遍历过程如表 2-2 所示。

表 2-2 最佳优先遍历过程表

Todo 优先级队列	Visited 集合	Todo 优先级队列	Visited 集合
A	null	F,H	A,B,C,D,E
D,B,C,E,F	A	G,H	A,B,C,D,E,F
B,C,E,F	A,D	H	A,B,C,D,E,F,G
C,E,F	A,B,D	I	A,B,C,D,E,F,H,G
E,F	A,B,C,D	null	A,B,C,D,E,F,H,I



## 2.2 爬虫架构

本节首先介绍爬虫的基本架构，然后介绍可以在多台服务器上运行的分布式爬虫架构。

### 2.2.1 基本架构

一般的爬虫软件，通常都包含以下几个模块：

- 保存种子 URL 和待抓取 URL 的数据结构
- 保存已经抓取过的 URL 的数据结构
- 页面获取模块
- 对已经获取页面内容的各个部分进行抽取的模块，例如抽取 HTML、JavaScript 等

其他可选的模块包括：

- 负责连接前处理模块
- 负责连接后处理模块
- 过滤器模块
- 负责多线程的模块
- 负责分布式的模块

各模块详细介绍如下。

### 1. 保存种子和爬取出来的 URL 的数据结构

农民会把有生长潜力的籽用做种子，这里把一些活跃的网页用做种子 URL，例如网站的首页或者列表页，因为在这些页面中经常会发现新的链接。通常，爬虫都是从一系列的种子 URL 开始爬取，一般从数据库表或者配置文件中读取这些种子 URL。种子 URL 描述如表 2-3 所示。

表 2-3 最佳优先遍历过程表

字段名	字段类型	说明
Id	NUMBER(12)	唯一标识
url	Varchar(128)	网站 url
Source	Varchar(128)	网站来源描述
rank	NUMBER(12)	网站 PageRank 值

但是保存待抓取的 URL 数据结构因系统的规模和功能不同可能采用不同的策略。一个比较小的示例爬虫程序，可能就使用内存中的一个队列，或者是优先级队列进行存储。一个中等规模的爬虫程序，可能使用 BekerlyDB 这种内存数据库来存储，如果内存中存放不下的话，还可以序列化到磁盘上。但是，真正的大规模爬虫系统，是通过服务器集群来存储已经爬取出来的 URL 的，并且还会在存储 URL 的表中附带一些其他信息，比如说 PageRank 值等，供之后的计算使用。

### 2. 保存已经抓取过的 URL 的数据结构

已经抓取过的 URL 的规模和待抓取的 URL 的规模是一个相当的量级，正如我们前面介绍的 Todo 表和 Visited 表。它们唯一的区别是，Visited 表会经常被查询，以便确定发现的 URL 是否已经处理过。因此，Visited 表数据结构如果是一个内存数据结构的话，可以采用 Hash（HashMap 或者 HashSet）来存储，如果保存在数据库中的话，可以对 URL 列建立索引。

### 3. 页面获取模块

当从种子 URL 队列或者抓取出来的 URL 队列中获得 URL 后，便要根据这个 URL 来获得当前页面的内容，方法非常简单，就是普通的 I/O 操作。在这个模块中，仅仅是把 URL 所指的内容按照二进制的格式读出来，而不对内容做任何处理。

### 4. 提取已经获取网页内容中的有效信息

从页面获取模块的结果是一个表示 HTML 源代码的字符串。从这个字符串中抽取各种





相关的内容是爬虫软件的目的。因此，这个模块就显得非常重要。

通常在一个网页中，除了包含文本内容还有图片、超链接等。对于文本内容，首先把 HTML 源代码的字符串保存成 HTML 文件即可。关于超链接提取，可以根据 HTML 语法，使用正则表达式来提取，并且把提取的超链接加入到 Todo 表中，也可以使用专门的 HTML 文档解析工具。

在网页中，超链接不光指向 HTML 页面，还会指向各种文件，对于除了 HTML 页面的超链接之外，其他内容的链接不能放入 Todo 表中，而要直接下载。因此在这个模块中，还必须包含提取图片、JavaScript、PDF、Doc 等内容，并且在提取过程中，还要针对 HTTP 协议来处理返回的状态码。本章我们主要研究网页的架构问题，下一章将会详细研究从各种文件格式提取有效信息。

#### 5. 负责连接前处理模块、负责连接后处理模块、过滤器模块

如果只抓取某个网站的网页，则可以对 URL 按域名过滤。

#### 6. 多线程模块

爬虫主要消耗三种资源：网络带宽、中央处理器和磁盘。三者中任何一者都有可能成为瓶颈，其中网络带宽一般是租用，所以价格相对昂贵。为了增加爬虫效率，最直接的方法就是使用多线程的方式进行处理。在爬虫系统中，将要处理的 URL 队列往往是唯一的，多个线程顺序地从队列中取得 URL，之后各自进行处理（处理阶段是并发进行）。通常可以利用线程池来管理线程。程序中可以使用的最大线程数是可配置的。

#### 7. 分布式处理

分布式是当今计算的主流，这项技术也可以同时用在网络爬虫上。

### 2.2.2 分布式爬虫架构

把抓取任务分布到不同的节点主要是为了可扩展性，也可以使用物理分布的爬虫系统，让每个爬虫节点抓取靠近它的网站。例如，北京的爬虫节点抓取北京的网站，上海的爬虫节点抓取上海的网站。又比如，电信网络中的爬虫节点抓取托管在电信的网站，联通网络中的爬虫节点抓取托管在联通的网站。

图 2-4 是一种没有中央服务器的分布式爬虫结构。

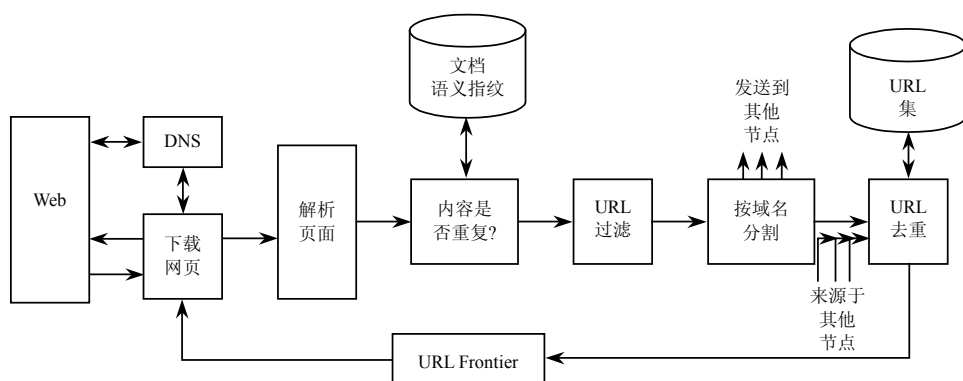


图 2-4 分布式爬虫结构图

要点在于按域名分配采集任务。每台机器扫描到的网址，不属于它自己的会交换给属于它的机器。例如，专门有一台机器抓取 s 开头的网站，如 <http://www.sina.com.cn> 和 <http://www.sohu.com>，而另外一台机器抓取 q 开头的网站，如 <http://www.qq.com>。

垂直信息分布式抓取的基本设计思路说明如下。

(1) 要处理的信息根据首字母散列到 26 个不同的爬虫服务器，让不同的机器抓取不同的信息。

(2) 每台机器通过配置文件读取自己要处理的字母。每台机器抓取完一条结果后把该结果写到统一的数据库中。比如说有 26 台机器，第一台机器抓取字母 a 开头的公司，第二台机器抓取字母 b 开头的公司，依次类推。

(3) 如果某一台机器抓取速度太慢，则把该任务拆分到其他的机器。

### 2.2.3 垂直爬虫架构

垂直爬虫往往抓取指定网站的新闻或论坛等信息。可以指定初始抓取的首页或者列表页，然后提取相关详细页中的有效信息存入数据库，总体结构如图 2-5 所示。

垂直爬虫涉及的功能有以下几点。

- 从首页提取不同栏目的列表页。
- 网页分类：把网页分类成列表页、详细页或未知类型。
- 列表页链接提取：从列表页提取同一个栏目下的列表页，这些页面往往用“下一页”、“尾页”等信息描述。

- 详细页面内容提取：从详细页提取网页标题、主要内容、发布时间等信息。

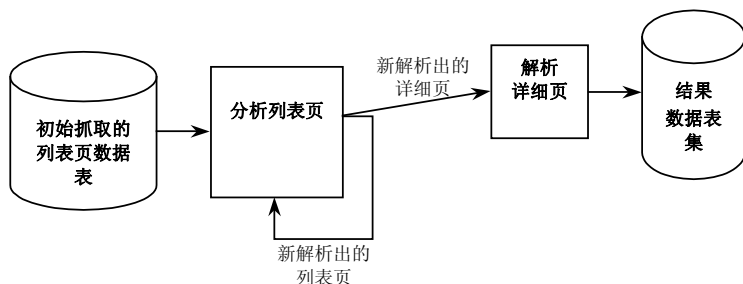


图 2-5 垂直爬虫结构图

可以每个网站用一个线程抓取，这样方便控制对被抓网站的访问频率。最好有通用的信息提取方式来解析网页，这样可以减少人工维护成本，同时也可以采用专门的提取类来处理数据量大的网站，这样可以提高抓取效率。



## 2.3 抓取网页

网络资源一般是 Web 服务器上的一些各种格式的文件。在 Java 中，一旦和被采集的 Web 服务器建立网络连接，对网络资源的操作就好像对本地文件的操作一样简单。通常，网络中使用 URL（统一资源定位符）来标示网络资源的位置。通过 URL，我们可以建立网络连接，并且获得相应的资源。在 Windows 平台下，当爬虫运行多日以后，由于操作系统对于进程废弃的网络连接回收不完全，导致爬虫出错，只有重启操作系统后才能继续运行爬虫，而在 Linux 下则不存在这样的问题。所以推荐在 Linux 环境下运行爬虫程序。

Java 中的 URL 类代表一个统一的资源定位符，资源可以是简单的文件或目录，也可以是对更为复杂对象的引用。

例如，`http://www.lietu.com/index.jsp` 是一个 URL 的例子。通常，URL 可分成几个部分。该 URL 示例指示使用的协议为超文本传输协议（http）并且该信息驻留在一台名为 `www.lietu.com` 的主机上，可以调用 URL 类的 `getHost()` 方法取得 URL 地址的主机名。主机上的信息名称为 `/index.jsp`。主机上此名称的准确含义取决于协议和主机。该信息一般存储在文件中，但可以随时生成。该 URL 的这一部分称为路径部分，可以调用 `getPath()` 方法取得路径部分。

需要通过 DNS 取得该 URL 域名的 IP 地址。在 Linux 下 DNS 解析的问题可以用 dig 或 nslookup 命令来分析，例如：

```
#dig www.lietu.com
#nslookup www.lietu.com
```

如果需要更换更好的 DNS 域名解析服务器，可以编辑 DNS 配置文件“/etc/resolv.conf”。DNS 解析是一个网络爬虫性能瓶颈。由于域名服务的分布式特点，DNS 可能需要多次请求转发，并在互联网上往返，需要几秒有时甚至更长的时间解析出 IP 地址。一个补救措施是引入 DNS 缓存，这样最近完成 DNS 查询的网址可能会在 DNS 缓存中找到，避免了访问互联网上的 DNS 服务器。JDK 1.6 内部有个 30 秒的 DNS 缓存，通过 `sun.net.InetAddress CachePolicy.get()` 方法可以查看缓存时间设置。Java 中要查找一个域名的 ip 最方便的办法就是调用 `java.net.InetAddress.getByName` (“www.lietu.com”)。

URL 可选择指定一个“端口”，它是用于建立到远程主机 TCP 连接的端口号。如果未指定该端口号，则使用协议默认的端口。例如，http 的默认端口为 80。还可以指定一个备用端口，如下所示：

```
http://www.lietu.com:80/index.jsp
```

### 2.3.1 下载网页的基本方法

在 Java 语言中，`java.net.URL` 类能够对实际的 URL 进行建模，通过这个类，可以对相应的 Web 服务器发出请求并且获得相应的文档。`java.net.URL` 类有一个默认的构造函数，使用 URL 地址作为参数，构造 URL 对象。

```
URL pageURL = new URL(path);
```

之后，可以通过获得的 URL 对象来取得网络流，进而像操作本地文件一样来操作网络资源。

```
InputStream stream = pageURL.openStream();
```

可以将网页看做网络文件，然后按照文件读取的方式把它读出来并保存到本地。以下是一个下载网页的小程序，简单地说明了网页下载的原理。

```
public class RetrivePage {
    public static String downloadPage(String path){
        //根据传入的路径构造 URL
    }
}
```



```
URL pageURL = new URL(path);
//创建网络流
BufferedReader reader = new BufferedReader(new InputStream
Reader(pageURL.openStream()));
String line;
//读取网页内容
StringBuilder pageBuffer = new StringBuilder();
while ((line = reader.readLine()) != null) {
    pageBuffer.append(line);
}
//返回网页内容
return pageBuffer.toString();
}
/**
 * 测试代码
 */
public static void main(String[] args) {
    //抓取 lietu 首页然后输出
    System.out.println(RetrivePage.downloadPage("http://www.lietu.
com"));
}
}
```

代码中的 `reader.readLine()` 有可能会抛出异常，因为网速可能不稳定，如果下载网页的过程中出现错误，还需要重试。如果重试仍然出错，下载线程可以调用 `sleep()` 方法休息片刻等待网速稳定后继续。

用 `Scanner` 对象下载网页的例子如下所示。

```
Scanner scanner = new Scanner(new InputStreamReader(
    pageURL.openStream(), "utf-8")); //指定编码格式 utf-8
scanner.useDelimiter("\\\\z"); //可以用正则表达式分段读取网页
//读取网页内容
StringBuilder pageBuffer = new StringBuilder();
while (scanner.hasNext()){
    pageBuffer.append( scanner.next() );
}
```

很多网页的编码格式是 `utf-8`，如果读入有乱码，则可以填入 `FireFox` 识别出的编码。

网络中的数据是通过 `TCP/IP` 协议来传输的。一般使用套接字来对 `TCP/IP` 协议编程。`URL` 对象的 `openStream` 方法使用了 `HTTP` 的 `GET` 命令返回 `Web` 页的正文内容。下面是一个直接使用套接字（`socket`）向 `Web` 服务器发送 `GET` 命令并输出返回结果的例子。

```

String host = "www.lietu.com"; //主机名
String file = "/index.jsp"; //网页路径
int port = 80; //端口号, 默认是 80

s = new Socket(host, port);

OutputStream out = s.getOutputStream();
PrintWriter outw = new PrintWriter(out, false);
outw.print("GET " + file + " HTTP/1.0\r\n");
outw.print("Accept: text/plain, text/html, text/*\r\n");
outw.print("\r\n");
outw.flush();//发送 GET 命令

InputStream in = s.getInputStream();
//InputStreamReader 的构造方法的第二个参数可以指定下载网页的编码格式
InputStreamReader inr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(inr);
String line;
while ((line = br.readLine()) != null){
    System.out.println(line); //输出返回的网页
}

```

Web 服务器返回的结果除了网页内容, 在此之前还包括头域 (header field) 信息, 如下所示:

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=UTF-8
Content-Length: 5367
Date: Tue, 29 Jun 2010 00:32:12 GMT
Connection: close

```

网页的编码方式由 Content-Encoding 或 Content-Type 定义, 它的长度由 Content-Length 或 Content-Range 定义。这些信息都可以通过 HttpURLConnection 得到。表 2-4 介绍了 HttpURLConnection 中的方法。

表 2-4 HttpURLConnection 中的方法

方 法	描 述
getHeaderField	取得头域信息
getContentType	取得网页类型
getLastModified	取得网页修改时间, 如果没有则返回 0
getContentEncoding	取得编码类型
getContentLength	取得网页长度



`getContentType` 等方法的错误之处在于只识别头信息中小写的字符串，需要修改成大小写不敏感。

```
public String getHeaderField(String fieldKey) throws IOException {
    URLConnection con = url.openConnection();
    header = con.getHeaderFields();

    Iterator i = getHeaderFields().keySet().iterator();
    String key = null;
    while (i.hasNext()) {
        key = (String) i.next();
        if (key == null) {
            if (fieldKey == null) {
                return (String) ((List) (getHeaderFields().get(null))).get(0);
            }
        } else {
            if (key.equalsIgnoreCase(fieldKey)) {
                return (String) ((List) (getHeaderFields().get(key))).get(0);
            }
        }
    }
    return null;
}
```

在实际的项目中，要处理各种复杂的网络环境。例如，需要处理 HTTP 返回的状态码，设置 HTTP 代理，处理 HTTPS 协议，设置 Cookie 等。尽管 `java.net` 包提供了通过 HTTP 访问资源的基本功能，但它没有提供全面的灵活性和套接字连接池等开发爬虫需要的功能。为了便于应用程序的开发，实际开发时常常使用开源项目 `HttpClient`。可以从 <http://hc.apache.org/downloads.cgi> 下载 `HttpClient` 最新的 4.2 版本。`HttpClient` 的 JavaDoc API 说明文档可以在 `httpcomponents-core-4.2-bin.zip` 中找到。

`HttpClient` 支持所有定义在 HTTP/1.1 版本中的 HTTP 方法。对于每个方法类型都有一个特定的类，爬虫最常用的是表示 HTTP GET 方法的 `org.apache.http.client.methods.HttpGet`。这样是为了避免误抓登录后才能看到的数据。爬虫开发中还可能用到 `HttpHead` 或 `HttpPost`。

使用 `HttpClient` 获取网页内容的代码如下：

```
//创建一个客户端，类似于打开一个浏览器
DefaultHttpClient httpClient = new DefaultHttpClient();

//创建一个 GET 方法，类似于在浏览器地址栏中输入一个地址
```

```

HttpGet httpget = new HttpGet("http://www.lietu.com/");

//类似于在浏览器地址栏中输入回车，获得网页内容
HttpResponse response = httpClient.execute(httpget);

//查看返回的内容，类似于在浏览器中查看网页源代码
HttpEntity entity = response.getEntity();
if (entity != null) {
    //读入内容流，并以字符串形式返回，这里指定网页编码是 UTF-8
    System.out.println(EntityUtils.toString(entity, "utf-8"));
    //网页的 Meta 标签中指定了编码
    EntityUtils.consume(entity); //关闭内容流
}

//释放连接
httpClient.getConnectionManager().shutdown();

```

### 2.3.2 网页更新

经常有人会问：“有没有什么新消息？”这说明人的大脑是增量获取信息的，对爬虫来说也是如此。网站中的内容经常会变化，这些变化经常在网站首页或者目录页中出现。为了提高采集效率，往往考虑增量采集网页。可以把这个问题看成是被采集的 Web 服务器和存储库同步的问题。更新网页内容的基本原理是：下载网页时，记录网页下载时的时间；增量采集时，判断 URL 地址对应的网页是否有更新。

泊松过程是指一种累计随机事件发生次数的最基本的独立增量过程。例如，随着时间增长累计某电话交换台收到的呼叫次数就构成一个泊松过程。网页更新过程符合泊松过程，网页更新时间间隔符合泊松指数分布。对于不同类型的网站采用不同的更新策略。例如.com 域名的网站更新频率较高，而.gov 域名的网站更新频率较低。

对于一些不太可能会更新的网页，只抓取一遍即可。但有些网页，例如首页或者列表，其页更新频率较高，所以需要隔一段时间就检测这些网页是否更新。如果只是想看看网页是否有更新，可以用 HTTP 的 HEAD 命令查看网页的最后修改时间。

```

String host = "www.lietu.com"; //主机名
String file = "/index.jsp"; //网页路径
int port = 80; //端口号
Socket s = new Socket(host, port); //建立套接字对象

OutputStream out = s.getOutputStream();
PrintWriter outw = new PrintWriter(out, false);

```





```
outw.print("HEAD " + file + " HTTP/1.0\r\n");//发送 HEAD 命令
outw.print("Accept: text/plain, text/html, text/*\r\n");
outw.print("\r\n");
outw.flush();

InputStream in = s.getInputStream();//返回头信息
InputStreamReader inr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(inr);
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
```

上面的程序在控制台打印结果如下所示：

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
ETag: W/"6810-1268491592000"
Last-Modified: Sat, 13 Mar 2010 14:46:32 GMT
Content-Type: text/html
Content-Length: 6810
Date: Tue, 15 Jun 2010 01:48:27 GMT
Connection: close
```

上述输出结果中“Last-Modified”行记录了网页的最后修改时间。“Date”行返回的是 Web 服务器的当前时间。可以通过 `URLConnection` 对象取得网页的修改时间，代码如下所示：

```
URL u = new URL("http://www.lietu.com");
URLConnection http = (URLConnection) u.openConnection();
http.setRequestMethod("HEAD");
System.out.println(u + " 更新时间 " + new Date(http.getLastModified()));
```

有些网页头信息没有包括更新时间，这时候可以通过判断网页长度来检测网页是否有更新。当然也可能网页更新了，但是长度没变，但实际上，这种可能性非常小。

日常生活中，有人会问：“今天有没有什么新消息？”所以有时候我们会根据时间来判断是否有新信息。条件下载命令可以根据时间条件下载网页。再次请求已经抓取过的页面时，爬虫往 Web 服务器发送 If-Modified-Since 请求头，其中包含的时间是先前服务器端发过来的 Last-Modified 最后修改时间戳。这样让 Web 服务器端进行验证，通过这个时间戳判断爬虫上次抓过的页面是否有修改。如果有修改，则返回 HTTP 状态码 200 和新的内容。如果没有变化，则只返回 HTTP 状态码 304，告诉爬虫页面没有变化。这样可以大大减少

在网络上传输的数据，同时也减轻了被抓取服务器的负担。下面的代码通过套接字发送 If-Modified-Since 头信息。

```
outw.print("GET " + file + " HTTP/1.0\r\n");
outw.print("If-Modified-Since: Thu, 01 Jul 2011 07:24:54 GMT\r\n");
```

HTTP/1.1 中还有一个 Etag 可以用来判断请求的文件是否被修改。可以把 Etag 看成网页的版本标志。Etag 主要为了解决 Last-Modified 无法解决的一些问题。

(1) 一些文件也许会周期性地更改，但是它的内容并不改变（仅仅改变了修改时间），这时我们并不希望客户端认为这个文件被修改了而重新下载。

(2) 某些文件修改得非常频繁，比如在秒以下的时间内进行修改（比方说 1 秒内修改了  $N$  次），If-Modified-Since 能检查到的粒度是秒级的，无法判断这种修改。

(3) 不能精确地得到某些 Web 服务器文件的最后修改时间。

在第一次抓取时记录网页的 Etag。下次发起 HTTP GET 请求一个文件，同时发送一个 If-None-Match 头，这个头的内容就是我们第一次请求时 Web 服务器返回的 Etag：6810-1268491592000。如果已经修改，则返回 HTTP 状态码 200 和新的内容。如果没有修改，则 If-None-Match 为 False，返回 HTTP 状态码 304。例如：

```
outw.print("GET " + file + " HTTP/1.0\r\n");
outw.print("If-None-Match: \"1272af65f918cb1:84f\" \r\n");
```

可以用 Range 条件下载部分网页。比如某网页的大小是 1000 字节，爬虫请求这个网页时用 “Range: bytes=0-500”，那么 Web 服务器应该把这个网页开头的 501 个字节发回给爬虫。

### 2.3.3 抓取限制应对方法

有些网站对于同一个 IP 在一段时间内的访问次数有限制。可以使用 Socket 代理来更改请求的 IP。这时可以通过大量不同的 Socket 代理循环访问网站。

首先建立有效代理列表 proxyIP.txt：

```
219.93.178.162:3128
222.135.79.253:8080
203.160.1.38:554
132.239.17.225:3124
```



```
169.229.50.5:3124
203.160.1.146:554
203.160.1.49:554
```

有些可以自动发现有效代理的软件，例如“花刺代理”等。然后建立 ProxyDB 类，循环使用这些 Socket 代理：

```
int pos = proxyIpList.get(count).toString().indexOf(":");
if (pos > 0) {
    String port = proxyIpList.get(count).toString().substring(pos+1);
    proxyAddr = proxyIpList.get(count).toString().substring(0,pos);
    proxyPort = Integer.parseInt(port);
    SocketAddress socketaddress = new InetSocketAddress(proxyAddr,
    proxyPort);
    proxy = new Proxy(Proxy.Type.HTTP,socketaddress);
}
```

最后通过下载网页 URL 的 openConnection 方法使用它：

```
url.openConnection(ProxyDB.getProxy());
```

因为 IP 地址资源很宝贵，大部分用户都是通过动态 IP 地址上网的。所谓动态就是指当你每一次上网时，电信或网通会随机给你分配一个 IP 地址。断网后重新上网就可以更换 IP 地址，使用新的 IP 地址继续抓取信息。使用浏览器 Finefox 下的插件浏览器 Firefox 下的插件 Firebug 分析出点击“重启路由器”时，浏览器向路由器发送的请求内容如下所示：

```
GET /userRpm/SysRebootRpm.htm?Reboot=%D6%D8%C6%F4%C2%B7%D3%C9%C6%F7
HTTP/1.1
Host:192.168.1.1
Authorization:Basic YWRtaW46YWRtaW4=
```

其中，Authorization 请求头的内容中，“Basic”表示“Basic authorization 验证”；“YWRtaW46YWRtaW4=”是使用 Base64 编码后的用户名和密码，解密后是“admin:admin”。

同样，也可以用程序实现以上的 HTTP 请求：

```
//modem 的用户名和密码
String data = "admin:admin";
String authorization = Base64.encode(data.getBytes());

String host = "192.168.1.1"; //modem 的 ip 地址
String file =
```

```

"/userRpm/SysRebootRpm.htm?Reboot=%D6%D8%C6%F4%C2%B7%D3%C9%C6%F7";
//网页路径
int port = 80; //端口号

Socket s = new Socket(host, port);

OutputStream out = s.getOutputStream();
PrintWriter outw = new PrintWriter(out, false);
outw.print("GET " + file + " HTTP/1.1\r\n");
outw.print("Authorization:Basic "+authorization+"\r\n");
outw.print("\r\n");
outw.flush();

```

可以发送和浏览器类似的头信息来简单地模仿浏览器访问，示例代码如下所示：

```

URL pageURL = new URL(path);
URLConnection uc = (URLConnection)pageURL.openConnection();
uc.setRequestProperty("User-Agent", "Internet Explorer");//发送头信息
uc.setRequestProperty("Host", pageURL.getHost());
uc.setRequestProperty("Accept-Language", "zh-cn");
uc.setRequestProperty("Accept-Charset", "gb2312,utf-8;q=0.7,*;q=0.7n");
uc.connect();
InputStream urlStream = uc.getInputStream();
BufferedReader reader =
    new BufferedReader(new InputStreamReader(urlStream));

```

可以用 Firebug 查看 FireFox 浏览器发送的头信息，如图 2-6 所示。

<b>Host</b>	addons.mozilla.org
<b>User-Agent</b>	Mozilla/5.0 (Windows; U; Windows NT 6.1; zh-CN; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3
<b>Accept</b>	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
<b>Accept-Language</b>	zh-cn,zh;q=0.5
<b>Accept-Encoding</b>	gzip,deflate
<b>Accept-Charset</b>	GB2312,utf-8;q=0.7,*;q=0.7
<b>Keep-Alive</b>	115
<b>Connection</b>	keep-alive
<b>Referer</b>	http://www.google.com.hk/search?q=firebug&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:zh-CN:official&client=firefox-a
<b>Cache-Control</b>	max-age=0

图 2-6 FireFox 浏览器发送的头信息

有些登录后才能看到的信息可以人工登录后，手工在程序中设置动态的 Cookie 值。

```

//假设 Cookie 值在 cookieValue 中
uc.setRequestProperty("Cookie", cookieValue);

```

为了找出抓取中的问题，可以采用网络工具 wireshark 追踪。



### 2.3.4 URL 地址提取

网页中的 URL 地址可能是相对地址例如“./index.html”或者绝对地址。经常需要把相对地址转化为绝对地址。URL 对象的 `toString` 方法返回的是绝对地址，因此为了把相对地址转化成绝对地址，只需要生成相对地址对应的 URL 对象即可。`new URL(fromUrl, url)` 方法可以实现从相对地址 URL 和来源 URL 对象 `fromUrl` 生成新的 URL 对象。完整实例代码如下所示：

```
URL fromUrl = new URL("http://www.lietu.com");
Node h = node.getAttributes().getNamedItem("href");//取得节点的 href 属性
String url = h.getNodeValue();//获得超链接的值
url = (new URL(fromUrl, url)).toString();//转成 String 类型
```

有些网址是通过提交表单，或者通过 JavaScript 来跳转的。可以参考 `HtmlUnit` (<http://htmlunit.sourceforge.net/>) 中的相关实现来获取。

### 2.3.5 抓取 JavaScript 动态页面

很多网页中包含 JavaScript 代码。例如一些网页链接嵌入在 JavaScript 代码中，因此爬虫最好能够识别 JavaScript。虽然 JavaScript 从语义上看来和 Java 非常相近，但实际上 JavaScript 来自一个和 Java 完全不同的编程语言家族。JavaScript 是 Smalltalk 和 Lisp 语言的一个直系后裔。

Mozilla 发布了纯 Java 语言编写的 JavaScript 脚本解释引擎 Rhino(<http://www.mozilla.org/rhino/>)。Rhion 包含所有 JavaScript 1.7 的特征，并附带一个可以运行 JavaScript 脚本的 JavaScript 外壳。有一个 JavaScript 编译器把 JavaScript 源文件翻译成 Java 的 class 文件。有一个 JavaScript 调试器用于 Rhino 执行脚本。Rhino 可以实现从 Java 对象到动态页面脚本片段常用语言——JavaScript 对象的直接映射，这有利于简化脚本解析环境的构建工作，减少脚本解释引擎与脚本片段在实现语言方面的差异。同时，在脚本解析的过程中，Rhino 对于 JavaScript 对象的操作结果可以通过访问已经本地创建的、与其一一对应的 Java 对象直接获得，示例代码如下所示：

```
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("JavaScript");

String s = GetJSText.getAllJS().toString(); //读取 JavaScript 的文本
engine.eval(s); //加载 JavaScript

//engine.eval(new FileReader("HelloWorld.js"));//从文件读入 JavaScript 脚本
```

```
// Invocable inv = (Invocable) engine;  
// Object obj = inv.invokeFunction("getI"); //调用 JavaScript 函数 getI()  
// System.out.println(obj);
```

Rhino 的主要功能是脚本执行时的运行环境管理。运行环境是指用来保存所要执行脚本中的变量、对象和执行上下文的空间。运行环境中的变量和对象由运行环境内所有的执行上下文共享,即一个执行上下文创建的变量或对象其他上下文也可以访问,运行环境负责处理变量或对象访问时的同步和互斥问题。运行环境和执行上下文是执行脚本语句的场所,因此在应用程序中应首先调用 Rhino 提供的 API (应用程序接口) 建立一个运行环境和若干个执行上下文,然后调用相应的 API 建立脚本语言的内置对象。

HTML DOM 中只有 Window 和 Document 对象的方法参数中含有超链接网络地址信息和页面主体内容。因此在进行 HTML DOM 对象本地创建时,将其余对象的属性和方法简单地设置为空 (NULL)。

在 Window 和 Document 对象的方法参数中,与超链接网络地址、页面主体内容相关的函数可以分为两类。第一类以 Window 对象的 open 方法为代表,open 方法的参数是动态页面中的超链接网络地址,参数类型是 JavaScript 语言内置 String 类型。在引擎外创建该类方法时,声明该方法的行为是把参数,即超链接网络地址送入信息采集环节的待获取 URL 队列中。第二类以 Document 对象的 write 方法为代表,write 方法的参数(同为 JavaScript 语言内置 String 类型)是一段表达脚本片段最终在浏览器中呈现内容的静态网页源文件。类似于常见的静态网页,在作为 write 方法参数的网页源文件中,超链接网络地址和页面主体内容被分别以 URL 和文本信息的方式直接嵌入 HTML 标记中。在引擎外创建该类方法时,声明该方法的行为是把参数,即静态网页源文件写入位于本地的特定文件中。

由于 Rhino 能够自动在 Java 对象和 JavaScript 对象之间根据“对象名称一致性”的原则实现一一对应,因此当 Rhino 在执行脚本片断中的“Window.open()”与“Document.write()”时,实际上是分别调用在 Java 语言作用域中定义的、与这两个方法同名的 Java 函数,执行函数体中关于函数行为的描述。

在完成脚本片断提取和 HTML DOM 本地创建后,就可以调用 Rhino 提取 JavaScript 动态页面中的超链接网络地址及页面主体内容。当遇到脚本片段中的 HTML DOM 时,Rhino 根据引擎外创建的同名函数体中的行为描述执行相应动作。

根据 HTML DOM 本地创建结果,Rhino 将脚本片段中 Window 对象方法 open 参数



体现的超链接网络地址直接送入信息采集环节的待获取 URL 队列中，实现动态页面内含超链接的递归获取功能。与其类似，把脚本片段中 Document 对象方法 write 参数所指向的静态网页源文件写入本地特定的文件中。在此基础上，使用传统的 HTML 标记识别方法，提取得到的静态网页源文件中的超链接网络地址与页面主体内容，将前者送入信息采集环节的 URL 队列，把后者交到信息采集环节统一实现数据存储。下面的网页按钮上存在 JavaScript 形式的跳转链接：

```
<html>
<head>
<script language="javascript">
function redirectNow(){
window.location.href="/AboutUs.htm";
}
</script>
</head>
<body>
<input type="button" value="Redirect using location.href" onclick="
redirectNow()">
</body>
</html>
```

提取这个跳转链接可以采用 HtmlUnit (<http://htmlunit.sourceforge.net/>) 实现。HtmlUnit 底层也是调用了 Rhino，但是绕过了 Rhino 的一些错误，主要代码如下所示：

```
public static void main(String[] args) throws Exception {
    WebClient webClient = new WebClient();
    String url = "http://www.lietu.com/lab/location.html";
    HtmlPage page = (HtmlPage) webClient.getPage(url);
    DomNodeList nodeList = page.getChildNodes();
    for (Object node : nodeList) {
        HTMLElement n = (HTMLElement) node;
        extract(n);
    }
}

public static void extract(HTMLElement node)
    throws IOException {
    if ("input".equals(node.getNodeName())) {
        HtmlInput inputHtml = (HtmlInput) node;
        HtmlPage newPage = inputHtml.click();
        System.out.println(newPage.toString()); // 按钮跳转的新页面
    }
}
```

```

DomNode childNode = node.getFirstChild();

if (childNode instanceof HTMLElement) {
    HTMLElement child = (HTMLElement) childNode;
    while (child != null) {
        //递归调用 extract 方法
        if(child == childNode)    {
            extract(child);
        }
        childNode = childNode.getNextSibling();
        if (childNode == null) {
            break;
        }
        if (childNode instanceof HTMLElement) {
            child = (HTMLElement) childNode;
        }
    }
}
}
}

```

但是 HtmlUnit 对于 JavaScript 框架 jQuery 的支持并不好。

另外一种方式是用 selenium (<http://seleniumhq.org>) 抓取 JavaScript 动态信息, 它有广泛的用户群和活跃的开发团队, Google 公司资助并且使用 Selenium。Selenium 的核心代码通过 JavaScript 完成, 可以运行在 FireFox 或 IE 等浏览器中。Watir (<http://watir.com/>) 是一个控制浏览器行为的类似项目。

如果只需要抓一个 Ajax 网站, 可以手工写模拟浏览器请求服务器端数据, 并提取信息的特定抓取代码。服务器端返回的数据往往是 JSON 格式的, 需要解析它。可以使用 GSON (<https://code.google.com/p/google-gson/>) 解析出想要的信息。

### 2.3.6 抓取即时信息

信息的即时性往往很重要。例如, 刚出现的经济信息可能在几秒钟以后就会影响相关股票的价格。

为了加快获取信息的速度, 可以先只抓取标题及 URL 地址。对于首页, 通过 HTTP 的 HEAD 命令判断页面是否已经更新。在不同的时间段, 用不同的频率来检查网页更新。

很多行业网站的首页是按板块组织的, 可以按板块提取 URL 地址。有的板块全部是新的, 也就是说, 新的信息从这个板块溢出了。对于从板块溢出的地址, 再从该版块对应的





索引页补全信息。查找一个板块对应索引页的方法是：查找该板块的“更多”链接对应的 URL 地址。

为了节省带宽，抓取到某个页面时，如果已经没有新的信息，可以不再继续往下检查这个页面的后续网页是否有更新。许多 Web 服务器具有发送压缩数据的能力，这可以将网络线路上传输的大量数据减少 60% 以上。

假设一个网站的首页有 90KB×8 位=720K 位，每秒需要同步 200 个网站，则需要大约 140MB 带宽。

### 2.3.7 抓取暗网

暗网的表现形式一般是：前台是一个表单来获取，提交后返回一个列表形式的搜索结果页，它们是由暗网后台数据库动态产生的。搜索结果页包含了指向详细内容页的链接。搜索引擎本身也可以看作是一个暗网。下面讨论抓取搜索引擎中的内容。

例如，有一个乡镇词表。“郭河”是一个小地名，但不知道它是一个镇还是一个村。把这个词提交给搜索引擎，返回的结果中可能包含了“郭河镇”这样的结果，这样就知道了“郭河”可能是一个镇。例如，把公司名称作为查询词提交给 Google 并从返回结果中提取联系方式：

```
String companyName = "SILK INDIA";
String searchWord = URLEncoder.encode(companyName, "utf-8");
//返回编码后的查询词
String searchURL = "http://www.google.com/search?q="+searchWord;
String content = RetrivePage.downloadPage(searchURL);
```

如果要提取公司的网址，可以提交“website:www 公司名”给 Google。但是对于同一个 IPGoogle 每天最多返回 1000 次查询。

以搜农网（www.sounong.net）为例，按照作物分类，将作物名称、供求类型等信息组装出查询 URL 地址，获得该 URL 的返回页面内容，匹配出当前查询的作物信息，然后计算出页面数量，这样组装 URL 时，动态替换页码进行翻页，具体代码如下所示：

```
final static String ORDER_FORMAT =
"http://www.sounong.net/newsounong/gq.jsp?q=%s&flag=show&adr=&sa%2F=
%B9%A9%C7%F3%CB%D1%CB%F7&flt=1&type=%C7%F3%B9%BA&sort=2";
String searchWord = URLEncoder.encode(keyword, "GBK");
String url = ORDER_FORMAT.replace("%s", searchWord);//执行参数替换
```

此外还可以抓取手机号码和所属地区的对应关系。提交一个手机号码到网站，返回所属地区。一般来说，前 7 位相同的手机都属于同一个地区，所以可以根据手机号码的前 7 位来判断地区。

对不同的语言，可以考虑抓取当地的搜索引擎网站，例如 Google 的葡萄牙文网站是 <http://www.google.pt>。

### 2.3.8 信息过滤

有时候可能需要按一个关键词列表来过滤信息，例如过滤黄色或其他非法信息。

调用 `indexOf` 方法来查找关键词集合看起来效率不高，Aho-Corasick 算法可以用来在文本中搜索多个关键词。当有一个关键词集合，想发现文本中所有出现关键词的位置，或者检查是否有关键词集合中的任何关键词出现在文本中时，这个实现代码是有用的。有很多不经常改变的关键词时，可以使用 Aho-Corasick 算法。其运行时间是输入文本的长度加上匹配关键词数目的线性。Aho-Corasick 算法根据两个发明人 Alfred V. Aho 和 Margaret J. Corasick 命名。

Aho-Corasick 算法把所有要查找的关键词构建成一个 Trie 树。Trie 树包含后缀树一样的链接集合，从代表字符串的每个节点（例如 abc）到最长的后缀（如果词典中存在 bc，则链接到 bc，否则如果词典中存在 c，则链接到 c，否则链接到根节点）。每个节点的失败匹配属性（failure）存储这个最长后缀节点。也就是说，Trie 树还包含一个从每个节点到存在于词典中的最长后缀节点链接。

假设词典中存在词：a、ab、bc、bca、c、caa。6 个词组成结构如图 2-7 所示的 Trie 树。

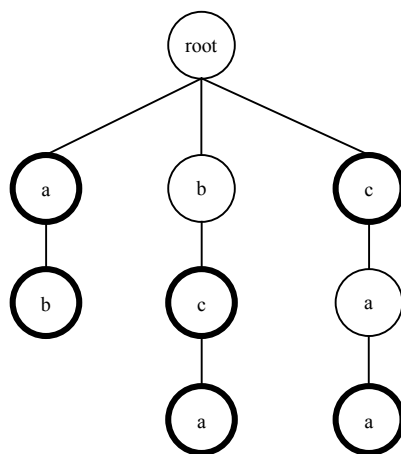


图 2-7 Trie 树结构

由指定字典构造的 Aho-Corasick 算法的数据结构如表 2-5 所示，表里面的每一行代表树中的一个节点，表里的列路径用从根到节点的唯一字符序列说明。

每取一个待匹配的字符后，当前的节点通过寻找它的孩子来匹配，如果孩子不存在，也就是匹配失败，则试图匹配该节点后缀的孩子，如果这样也没有匹配上，则匹配该节点后缀的后缀的孩子，最后如果什么也没有匹配上，就在根节点结束。

表 2-5 Trie 树结构

路 径	是否在字典里	后缀链接	词典后缀链接
()	否		
(a)	是	()	
(ab)	是	(b)	
(b)	否	()	
(bc)	是	(c)	(c)
(bca)	是	(ca)	(a)
(c)	是	()	
(ca)	否	(a)	(a)
(caa)	是	(a)	(a)

分析输入字符串“abccab”的匹配过程如表 2-6 所示。

表 2-6 Aho-Corasick 算法匹配过程

节 点	剩余的字符串	输出：结束位置	跳 转	输 出
()	abccab		从根节点开始	
(a)	bccab	a:1	从根节点()转移到孩子节点(a)	当前节点
(ab)	ccab	ab:2	节点(a)转移到孩子节点(ab)	当前节点
(bc)	cab	bc:3, c:3	节点(ab)转到后缀(b)，再跳转到孩子节点(bc)	当前节点，词典后缀节点
(c)	ab	c:4	节点(bc)转到后缀节点(c)，再跳转到根节点()，再跳转到孩子节点(c)	当前节点
(ca)	b	a:5	节点(c)跳转到孩子节点(ca)	词典后缀节点
(ab)		ab:6	节点(ca)跳转到后缀节点(a)，再跳转到孩子节点(ab)	当前节点

首先定义 Trie 树节点类：

```
private final class TreeNode {
    private char _char; // 节点代表的字符
    private TreeNode _parent; // 该节点的父节点
    private TreeNode _failure; // 匹配失败后跳转的节点
    private ArrayList<String> _results; // 存储模式串的数组变量
    private TreeNode[] _transitionsAr;
    // 存储孩子节点的哈希表
    private Hashtable<Character, TreeNode> _transHash;
    public TreeNode(TreeNode parent, char c) {
        _char = c;
        _parent = parent;
        _results = new ArrayList<String>(); // 存储所有没有重复的模式串的数组
    }
}
```

```

        _transitionsAr = new TreeNode[] {};
        _transHash = new Hashtable<Character, TreeNode>();
    }

    //将模式串中不在_results 中的模式串添加进来
    public void addResult(String result) {
        if (_results.contains(result))//如果已经包含该模式则不增加到模式串中
            return;
        _results.add(result);
    }

    public void addTransition(TreeNode node) { //增加一个孩子节点
        _transHash.put(node._char, node);
        TreeNode[] ar = new TreeNode[_transHash.size()];
        Iterator<TreeNode> it = _transHash.values().iterator();
        for (int i = 0; i < ar.length; i++) {
            if (it.hasNext()) {
                ar[i] = it.next();
            }
        }
        _transitionsAr = ar;
    }

    public TreeNode getTransition(char c) {
        return _transHash.get(c);
    }

    public boolean containsTransition(char c) {
        return getTransition(c) != null;
    }

    public char getChar() {
        return _char;
    }

    public TreeNode parent() {
        return _parent;
    }

    public TreeNode failure(TreeNode value) {
        _failure = value;
        return _failure;
    }

    public TreeNode[] transitions() {

```



```
        return _transitionsAr;
    }

    public ArrayList<String> results() {
        return _results;
    }
}
```

Trie 树的构建过程由构建树本身和增加失败匹配属性两步构成：

```
public StringSearch(String[] keywords) {
    buildTree(keywords); //构建树
    addFailure(); //增加失败匹配属性
}
```

构建树的过程：

```
void buildTree() {
    _root = new TreeNode(null, ' ');

    for (String p : _keywords) {
        TreeNode nd = _root;

        for(char c : p.toCharArray()){
            TreeNode ndNew = null;
            for (TreeNode trans : nd.transitions())
                if (trans.getChar() == c) {
                    ndNew = trans;
                    break;
                }

            if (ndNew == null) {
                ndNew = new TreeNode(nd, c);
                nd.addTransition(ndNew);
            }
            nd = ndNew;
        }
        nd.addResult(p);
    }
}
```

增加失败匹配属性的过程：

```
private void addFailure(){
    //所有词的第 n 个字节点的集合，n 从 2 开始
```

```

ArrayList<TreeNode> nodes = new ArrayList<TreeNode>();

//所有词的第2个字节节点的集合
for (TreeNode nd : _root.transitions()) {
    nd.failure(_root);
    for (TreeNode trans : nd.transitions()){
        nodes.add(trans);
    }
}

//所有词的第n+1个字节节点的集合
while (nodes.size() != 0) {
    ArrayList<TreeNode> newNodes = new ArrayList<TreeNode>();
    for (TreeNode nd : nodes) {
        TreeNode r = nd.parent()._failure;
        char c = nd.getChar();

        //如果在父节点的失败节点的孩子节点中没有同样字符结尾的
        //则继续在失败节点的失败节点中找
        while (r != null && !r.containsTransition(c))
            r = r._failure;
        if (r == null)
            nd._failure = _root;
        else {
            nd._failure = r.getTransition(c);
            for (String result : nd._failure.results()) {
                nd.addResult(result);
            }
        }

        for (TreeNode child : nd.transitions()){
            newNodes.add(child);
        }
    }
    nodes = newNodes;
}
_root._failure = _root;
}

```

为了加深理解，可以打印生成的 Trie 树。遍历二叉树的方法有：先序遍历、后序遍历、中序遍历。深度遍历、广度遍历是针对普通树。因为这是一棵普通的树，所以采用深度遍历的方式打印树结构。

```

//深度遍历的递归函数
public void depthSearch(TreeNode node, StringBuilder ret,int deapth) {

```



```
        if (node != null) {
            for (int i = 0; i < deapth; i++)
                ret.append("| ");
            ret.append("|—");
            ret.append(node._char + "\n");
            for (TreeNode child : node.transitions()) {
                int childDeapth = deapth + 1; //计算深度并赋值
                depthSearch(child, ret, childDeapth);
            }
        }
    }

    //打印树节点
    public String toString() {
        StringBuilder ret = new StringBuilder();
        ret.append("打印树节点: \n");
        int deapth = 0;
        depthSearch(_root, ret, deapth);
        return ret.toString();
    }
}
```

从输入文本查找关键词集合的过程：

```
public StringSearchResult[] findAll(String text) {
    ArrayList<StringSearchResult> ret = new ArrayList<StringSearchResult>();
    TreeNode ptr = _root;
    int index = 0;

    while (index < text.length()) {
        TreeNode trans = null;
        while (trans == null) {
            trans = ptr.getTransition(text.charAt(index));

            if (ptr == _root)
                break;
            if (trans == null) {
                ptr = ptr._failure;
            }
        }
        if (trans != null)
            ptr = trans;
        //增加找到的每一个词到结果中
        for (String found : ptr.results())
            ret.add(new StringSearchResult(index - found.length() + 1, found));
    }
}
```

```

        index++;
    }

    return ret.toArray(new StringSearchResult[ret.size()]);
}

```

### 2.3.9 最好优先遍历

下面实现一个 Best-First 的爬虫，也就是说每次取的 URL 地址都是 Todo 列表中最好的 URL 地址。为了实现快速查找和操作，一般使用 Berkeley DB 来实现 Todo 列表。

Berkeley DB 中存储的一般是一个关键字和值的 Map。为了同时实现按照 URL 地址查找和分值排序，简单的 map 不能满足要求，这时可以使用 SecondaryIndex 和 PrimaryIndex 来达到多列索引的目的。实现持久化的基本类如下所示：

```

@Entity
public class NewsSource {
    @PrimaryKey public String URL;
    public String source;
    public int level;
    public int rank;
    public String urlDesc = null;
    @SecondaryKey(related=Relationship.MANY_TO_ONE) public int score;
}

```

Todo 列表的构造方法如下：

```

public TodoTaskList(Environment env) throws Exception {
    StoreConfig storeConfig = new StoreConfig();
    storeConfig.setAllowCreate(true);
    storeConfig.setTransactional(false);
    store = new EntityStore(env, "classDb", storeConfig);
    newsByURL = store.getPrimaryIndex(String.class, NewsSource.class);
    secondaryIndex =
        store.getSecondaryIndex(this.newsByURL, Integer.class,
            "score");
}

```

从 Todo 列表取得最大分值的 URL 地址的方法如下所示：

```

public NewsSource removeBest() throws DatabaseException {
    Integer score = secondaryIndex.sortedMap().lastKey();
    if (score != null){
        EntityIndex<String,NewsSource> urlLists = secondaryIndex.

```





```
        subIndex(score);
        EntityCursor<String> ec = urlLists.keys();
        String url = ec.first();
        ec.close();
        NewsSource source = urlLists.get(url);
        urlLists.delete(url);
        return source;
    }
    return null;
}
```



## 2.4 存储 URL 地址

在科技论文正式发表前，为了避免重复研究和抄袭，需要到专门的科技情报所做论文查新。为了避免重复抓取网页，URL 地址也需要查新。判断解析出的 URL 是否已经遍历过，叫做 URLSeen 测试。URLSeen 测试对爬虫性能有重要影响。本节介绍两种实现快速 URLSeen 测试的方法。

在介绍爬虫架构的时候，我们讲解了 Frontier 组件的作用。它作为一个基础组件，为爬虫提供 URL。因此，在 Frontier 中有一个数据结构来存储 URL。在一些小的爬虫程序中，使用内存队列（ArrayList、HashMap 或 Queue）或者优先级队列来存储 URL，但内存是有限的。通常在商业应用中，URL 地址的数据量非常大。早期的爬虫经常把 URL 地址放在数据库表中，但数据库对于这种简单的结构化存储来说效率太低，因此考虑使用内存数据库来存储。BerkeleyDB 就是一种常用的内存数据库。

### 2.4.1 BerkeleyDB

BerkeleyDB (<http://www.oracle.com/database/berkeley-db/index.html>) 是一个嵌入式数据库。不是说它只应用在嵌入式系统上，嵌入式数据库的意思是不需要通过 JDBC 访问数据库，也不单独启动进程来管理数据。BerkeleyDB 中的一个数据库只能存储一些键/值对，也就是键和值两列。BerkeleyDB 底层实现采用 B 树，可以把它看成可以存储大量数据的 HashMap。BerkeleyDB 的 C++ 版本首先出现，在此基础上又实现了 Java 本地版本，但本书中只用到 Java 版本。可以用它来存储已经抓取过的 URL 地址。如果使用 BerkeleyDB Java Edition 4.0.103，在 Eclipse 项目中只需要导入一个 jar 包——je-4.0.103.jar。在 BerkeleyDB 中，数据库存储在环境变量中，所以首先要新建环境变量：

```
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setTransactional(false); //不需要事务功能
envConfig.setAllowCreate(true); //允许创建新的数据库文件
exampleEnv = new Environment(envDir, envConfig);
```

释放环境变量的方法如下所示：

```
exampleEnv.sync(); //同步内存中的数据到硬盘
exampleEnv.close(); //关闭环境变量
exampleEnv = null;
```

创建数据库，键是字符串，值是一个类：

```
String databaseName= "ToDoTaskList.db";
DatabaseConfig dbConfig = new DatabaseConfig();
dbConfig.setAllowCreate(true); //允许创建新的数据库文件
dbConfig.setTransactional(false); //不需要事务功能

//打开用来存储类信息的数据库
dbConfig.setSortedDuplicates(false);
//用来存储类信息的数据库不要求能够存储重复的关键字
Database myClassDb = exampleEnv.openDatabase(null, "classDb",
dbConfig);
//初始化用来存储序列化对象的 catalog 类
catalog = new StoredClassCatalog(myClassDb);
TupleBinding keyBinding =
    TupleBinding.getPrimitiveBinding(String.class);
//把 value 作为对象的序列化方式存储
SerialBinding valueBinding = new SerialBinding(catalog, NewsSource.
class);
store = exampleEnv.openDatabase(null, databaseName, dbConfig);
store.close(); //关闭存储数据库
myClassDb.close(); //关闭元数据库
```

建立数据的映射：

```
//创建数据存储的映射视图
this.map = new StoredSortedMap(store, keyBinding, valueBinding, true);
```

然后可以像操作普通的哈希表一样，对 `StoredSortedMap` 用 `put` 方法写入数据，用 `get` 方法读出数据，通过迭代器遍历访问。但是普通的哈希表只能把所有的数据存储在内存中，而 `StoredSortedMap` 则会通过内外存交换支持更多的数据存取。完整的例子如下所示：



```
String dir = "./db/";
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setTransactional(false);
envConfig.setAllowCreate(true);

Environment env = new Environment(new File(dir), envConfig);

//使用一个通用的数据库配置
DatabaseConfig dbConfig = new DatabaseConfig();
dbConfig.setTransactional(false);

dbConfig.setAllowCreate(true);
//如果有序列化绑定则需要类别数据库
Database catalogDb = env.openDatabase(null, "catalog", dbConfig);
ClassCatalog catalog = new StoredClassCatalog(catalogDb);

//关键字数据类型是字符串
TupleBinding<String> keyBinding =
    TupleBinding.getPrimitiveBinding(String.class);

//值数据类型也是字符串
SerialBinding<String> dataBinding =
    new SerialBinding<String>(catalog, String.class);

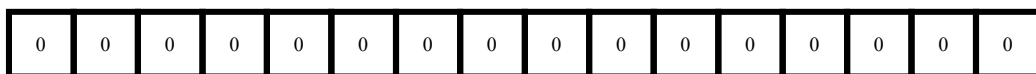
Database db = env.openDatabase(null, "url", dbConfig);

//创建一个映射
SortedMap<String, String> map = new StoredSortedMap<String, String>
    (db, keyBinding, dataBinding, true);
//把抓取过的 URL 地址作为关键字放入映射
String url = "http://www.lietu.com";
map.put(url, null);
if (map.containsKey(url)) {
    System.out.println("已抓取");
}
```

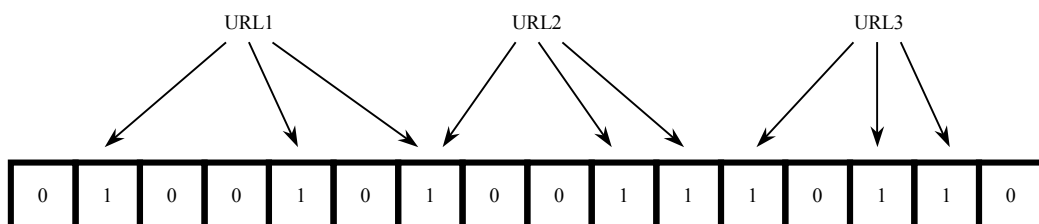
为了能正确处理大量数据，需要增加 Java 虚拟机运行的内存使用量，否则会出现内存溢出的错误。例如将最大内存用量增加到 800MB，可以使用虚拟机参数“-Xmx800m”。

### 2.4.2 布隆过滤器

判断 URL 地址是否已经抓取过还可以借助于布隆过滤器（Bloom Filter）。布隆过滤器的实现方法是：利用内存中的一个长度是  $m$  的位数组  $B$ ，对其中所有位都置 0，如图 2-8 所示。

图 2-8 位数组  $B$  的初始状态

然后对每个遍历过的 URL 根据  $k$  个不同的散列函数执行散列，每次散列的结果都是不大于  $m$  的一个整数  $a$ 。根据散列得到的数在位数组  $B$  对应的位上置 1，也就是让  $B[a]=1$ 。图 2-9 显示了放入 3 个 URL 后位数组  $B$  的状态，这里  $k=3$ 。

图 2-9 放入数据后位数组  $B$  的状态

每次插入一个 URL，执行  $k$  次散列，只有当全部位都已经置 1 了才认为这个 URL 已经遍历过。如下是使用布隆过滤器（SimpleBloomFilter）的一个例子：

```
//创建一个 100 位的布隆过滤器，优化成包含 4 个项目
SimpleBloomFilter urlSeen = new SimpleBloomFilter(100, 4);
//增加内容到布隆过滤器
urlSeen.add("www.lietu.com");
urlSeen.add("www.sina.com");
urlSeen.add("www.qq.com");
urlSeen.add("www.sohu.com");
//测试布隆过滤器是否记得这个项目
if (urlSeen.contains("www.lietu.com")) {
    System.out.println("已经存在的概率 "
        + (1 - urlSeen.expectedFalsePositiveProbability()));
} else {
    System.out.println("一定不存在");
}
```

布隆过滤器如果返回不包含某个项目，那肯定就是没往里面增加过这个项目，如果返回包含某个项目，但其实可能没有增加过这个项目，所以有误判的可能。对爬虫来说，使用布隆过滤器的后果是可能导致漏抓网页。如果想知道需要使用多少位才能降低错误概率，可以从表 2-7 的存储项目和位数比率估计布隆过滤器的误判率。

表 2-7 布隆过滤器误判率表

比率 (items:bits)	误 判 率	比率 (items:bits)	误 判 率
1:1	0.63212055882856	1:16	0.00046557303372
1:2	0.39957640089373	1:32	0.00000021167340
1:4	0.14689159766038	1:64	0.00000000000004
1:8	0.02157714146322		

为每个 URL 分配两个字节就可以达到千分之几的冲突。例如一个比较保守的实现，为每个 URL 分配了 4 个字节，项目和位之数比是 1:32，误判率是 0.00000021167340。对于 5000 万数量级的 URL，布隆过滤器只占用了 200MB 的空间，并且排重速度超快，一遍下来不到两分钟。

SimpleBloomFilter 中实现的把对象映射到位集合的方法代码如下所示：

```
public void add(E element) throws UnsupportedOperationException {
    long hash;
    String valString = element.toString();
    for (int x = 0; x < k; x++) {
        hash = createHash(valString + Integer.toString(x));
        hash = hash % (long)bitSetSize;
        bitset.set(Math.abs((int)hash), true);
    }
}
```

这个实现该方法计算了  $k$  个相互独立的散列值，因此误判率较低。

为了支持重新启动后运行，把布隆过滤器的状态保存到文件中：

```
public void saveBit(String filename) {
    File file = new File(filename);
    ObjectOutputStream oos =
        new ObjectOutputStream(new FileOutputStream(
            file, false));
    oos.writeObject(bitSet);
    oos.flush();
    oos.close();
}
```

如下代码把布隆过滤器的状态从先前保存的文件中读出来：

```
public void readBit(String filename) {
    File file = new File(filename);
```

```

        if (!file.exists()) {
            return;
        }
        bitSet.clear();
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(
            file));
        bitSet = (BitSet)ois.readObject();
        ois.close();
    }

```



## 2.5 并行抓取

单机并行抓取往往采用多线程同步 I/O 或者单线程异步 I/O。为了同时抓取多个网站的信息，可以考虑并行抓取。

在 Java 中，为了爬虫的稳定性，也可以用多线程来实现爬虫。一般情况下，爬虫程序需要在后台长期稳定运行。下载网页时，经常会出现异常。有些异常无法捕获，导致爬虫程序退出。为了主程序稳定，可以把下载程序放在子线程执行，这样即使子线程因为异常退出了，但是主线程并不会退出。测试代码如下所示：

```

public class MyThread extends Thread{
    public void run() {
        System.out.println("Throwing in " + "MyThread");
        throw new RuntimeException();
    }
}

public class ThreadTest {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        try {
            Thread.sleep(2000);
        }
        catch (Exception x) {
            System.out.println("Caught it");
        }
        System.out.println("Exiting main");
    }
}

```

### 2.5.1 多线程爬虫

利用多线程爬虫进行抓取是当前搜索引擎中普遍采用的架构模式，一个多线程爬虫架构如图 2-10 所示。

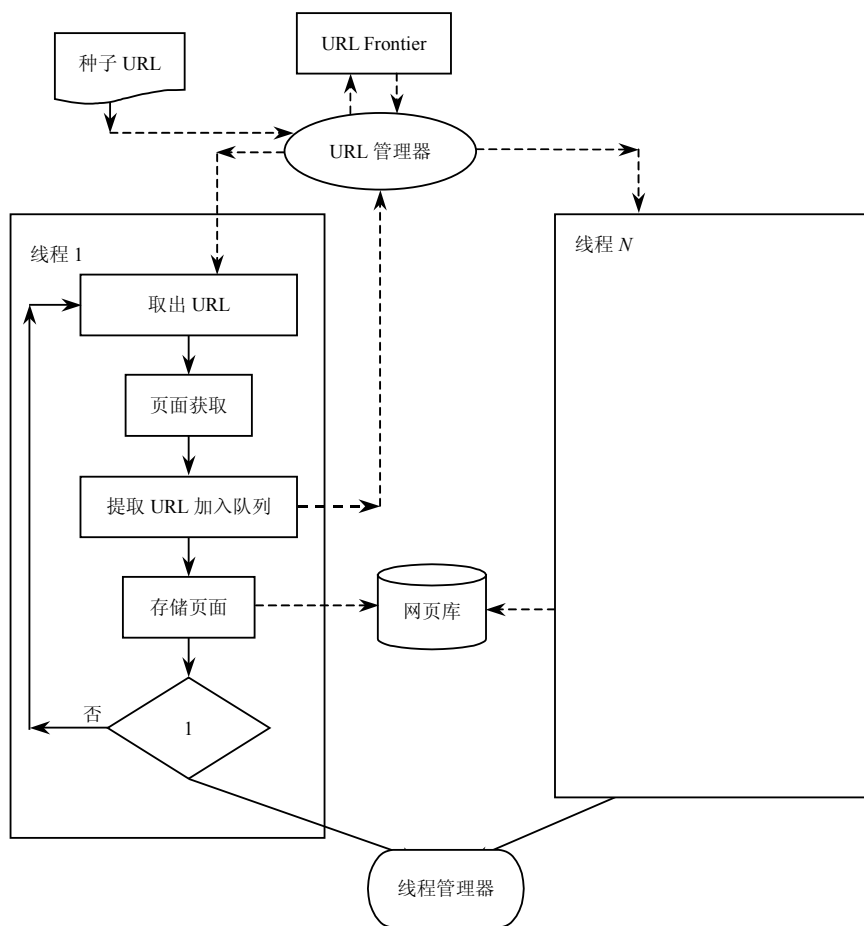


图 2-10 多线程爬虫架构

JDK 1.5 以后的版本提供了一个轻量级线程池 `ThreadPool`。可以使用线程池执行一组任务，最简单的任务不返回值给主调线程。要返回值的任务可以实现 `Callable<T>` 接口，线程池执行任务并通过 `Future<T>` 的实例获取返回值。

`Callable` 是类似于 `Runnable` 的接口，在其中定义可以在线程池中执行的任务。`Future` 表示异步计算的结果，它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。`Future` 的 `cancel` 方法取消任务的执行，`cancel` 方法有一个布尔参数，参数为 `true`

表示立即中断任务的执行，参数为 `false` 表示允许正在运行的任务运行完成。`Future` 的 `get` 方法等待计算完成，获取计算结果。

下面使用 `ThreadPool` 实现并行下载网页。在继承 `Callable` 方法的任务类中下载网页：

```
public class DownloadCall implements Callable<String> {
    private URL url;    //待下载的 url

    public DownloadCall(URL u) {
        url = u;
    }

    @Override
    public String call() throws Exception {
        String content = null;
        //下载网页
        return content;
    }
}
```

主线程类创建 `ThreadPool` 并执行下载任务的实现代码如下所示：

```
int threads = 4; //并发线程数量
ExecutorService es = Executors.newFixedThreadPool(threads); //创建线程池

Set<Future<String>> set = new HashSet<Future<String>>();

for (final URL url : urls) {
    DownloadCall task = new DownloadCall(url);
    Future<String[]> future = es.submit(task); //提交下载任务
    set.add(future);
}

//通过 future 对象取得结果，这一步不能省略，如果不需要返回值可以调用 Runnable
for (Future<String> future : set) {
    String content = future.get();
    //处理下载网页的结果
}
```

采用线程池可以充分利用多核 CPU 的计算能力，并且简化了多线程的实现。





## 2.5.2 垂直搜索的多线程爬虫

垂直行业网站一般不是太多，为了及时采集每个网站，可以每个线程抓取一个指定的网站，然后把抓取的信息直接写入索引。对 Lucene 来说，同一个时刻只能由一个线程写索引，而可以多个抓取线程同时并行下载。套用线程的生产者和消费者模型，这里索引线程是消费者，抓取线程是生产者。为了简化实现，以传递整数为例。

```
public class Indexer implements Runnable { //索引类
    private BlockingQueue<Integer> DataQueue;
    public Indexer(BlockingQueue<Integer> DataQueue) {
        this.DataQueue = DataQueue;
    }
    @Override
    public void run() {
        Integer i;
        while (!Thread.interrupted()){
            try {
                i = DataQueue.take();
                System.out.println("索引: " + i);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Spider implements Runnable { //爬虫类
    private BlockingQueue<Integer> DataQueue;
    private static int i = 0;
    public Spider(BlockingQueue<Integer> DataQueue) {
        this.DataQueue = DataQueue;
    }

    @Override
    public void run() {
        while (!Thread.interrupted()){
            try {
                DataQueue.add(new Integer(++i));
                System.out.println("生产: " + i);
                TimeUnit.MILLISECONDS.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

测试方法如下所示：

```
public static void main(String[] args) {
    BlockingQueue<Integer> DataQueue = new LinkedBlockingQueue<Integer>
        ();

    Thread pt = new Thread(new Spider(DataQueue)); //爬虫线程
    pt.start();

    Thread ct = new Thread(new Indexer(DataQueue)); //索引线程
    ct.start();
}
```

### 2.5.3 异步 I/O

对单线程并行抓取来说，异步（asynchronous）I/O 是很重要的基本功能。异步 I/O 模型大体上可以分为两种：反应式（Reactive）模型和前摄式（Proactive）模型。传统的 select/epoll/kqueue 模型，以及 Java NIO 模型，都是典型的反应式模型，即应用代码对 I/O 描述符进行注册，然后等待 I/O 事件。当某个或某些 I/O 描述符所对应的 I/O 设备上产生 I/O 事件（可读、可写、异常等）时，系统将发出通知，于是应用便有机会进行 I/O 操作并避免阻塞。由于在反应式模型中应用代码需要根据相应的事件类型采取不同的动作，最常见的结构便是嵌套的 if {...} else {...} 或 switch，并常常需要结合状态机来完成复杂的逻辑。前摄式模型则恰恰相反。在前摄式模型中，应用代码主动地投递异步操作而不管 I/O 设备当前是否可读或可写。投递的异步 I/O 操作被系统接管，应用代码也并不阻塞在该操作上，而是指定一个回调函数并继续自己的应用逻辑。当该异步操作完成时，系统将发起通知并调用应用代码指定的回调函数。在前摄式模型中，程序逻辑由各个回调函数串联起来：异步操作 A 的回调发起异步操作 B，B 的回调再发起异步操作 C，以此往复。

Java 6 版本开始引入的 NIO 包，通过 Selectors 提供了非阻塞式的 I/O。Java 7 附带的 NIO.2 文件系统中包含了异步 I/O 支持，也可以使用框架实现异步 I/O。

- Mina (<http://mina.apache.org/>) 为开发高性能和高可用性的网络应用程序提供了非常便利的框架。当前发行的 MINA 版本支持基于 Java 的 NIO 技术的 TCP/UDP 应用程序开发。MINA 是借由 Java 的 NIO 的反应式实现的模拟前摄式模型。
- Grizzly 是 Web 服务器 GlassFish 的 I/O 核心。Grizzly 还是一个独立于 GlassFish 的框架结构，可以单独用来扩展和构建自己的服务器软件。Grizzly 通过队列模型提供异步读/写。
- Netty (<http://www.jboss.org/netty>) 是一个 NIO 客户端服务器框架。

- Naga (<http://naga.googlecode.com>) 是一个很小的库，提供了一些 Java 类把普通的 Socket 和 ServerSocket 封装成支持 NIO 的形式。

从性能测试上比较，Netty 和 Grizzly 都很快，而 Mina 稍慢一些。

JDK 1.6 内部并不使用线程来实现非阻塞式 I/O。在 Windows 平台下使用 select(); 在新的 Linux 核下使用 epoll 工具。

Niocchi (<http://www.niocchi.com>) 是 Java 实现的开源异步 I/O 爬虫。

在爬虫中使用 NIO 的时候，主要用到下面两个类。

- java.nio.channels.Selector: Selector 类通过调用 select 方法，将注册的 Channel 中有事件发生的 SelectionKey 取出来进行处理。如果想要把管理权交到 Selector 类手中，首先先要在 Selector 对象中注册相应的 Channel。
- java.nio.channels.SocketChannel: SocketChannel 用于和 Web 服务器建立连接。

下面是使用 NIO 下载网页的例子，代码结构如图 2-11 所示。

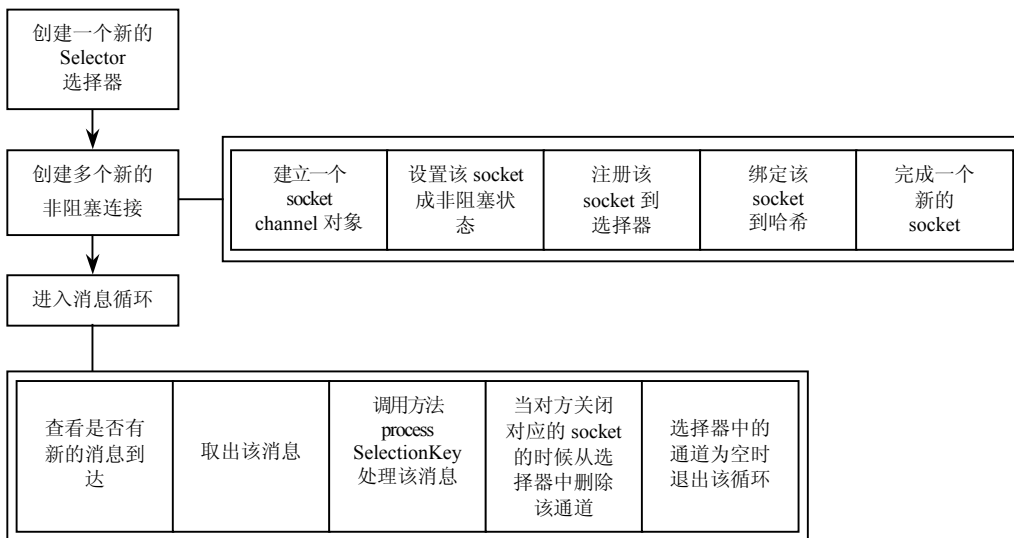


图 2-11 异步 I/O

使用 NIO 下载网页的具体实现代码如下所示：

```
public static Selector sel = null;
public static Map<SocketChannel, String> sc2Path = new HashMap<Socket
```

```

Channel, String>());

public static void setConnect(String ip, String path, int port) {
    try {
        SocketChannel client = SocketChannel.open();
        client.configureBlocking(false);
        client.connect(new InetSocketAddress(ip, port));
        client.register(sel, SelectionKey.OP_CONNECT | SelectionKey.
            OP_READ
                | SelectionKey.OP_WRITE); //
        sc2Path.put(client, path);
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
    }
}

public static void main(String args[]) {
    try {
        sel = Selector.open();
        setConnect("www.lietu.com", "/index.jsp", 80); //添加一个下载网址
        setConnect("hao123.com", "/book.htm", 80); //添加另一个下载网址

        while (!sel.keys().isEmpty()) {
            if (sel.select(100) > 0) {
                Iterator<SelectionKey> it = sel.selectedKeys().
                    iterator();
                while (it.hasNext()) {
                    SelectionKey key = it.next();
                    it.remove();
                    try {
                        processSelectionKey(key);
                    } catch (IOException e) {
                        key.cancel();
                    }
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(0);
    }
}

public static void processSelectionKey(SelectionKey selKey)
    throws IOException {
    SocketChannel sChannel = (SocketChannel) selKey.channel();
}

```



```
        if (selKey.isValid() && selKey.isConnectable()) {
            boolean success = sChannel.finishConnect();
            if (!success) {
                selKey.cancel();
            }
            sendMessage(sChannel, "GET " + sc2Path.get(sChannel)
                + " HTTP/1.0\r\nAccept: */*\r\n\r\n");
        } else if (selKey.isReadable()) {
            String ret = readMessage(sChannel);
            if (ret != null && ret.length() > 0) {
                System.out.println(ret);
            } else {
                selKey.cancel();
            }
        }
    }
}

//下载网页
public static String readMessage(SocketChannel client) {
    String result = null;
    ByteBuffer buf = ByteBuffer.allocate(1024);
    try {
        int i = client.read(buf);
        buf.flip();
        if (i != -1) {
            result = new String(buf.array(), 0, i);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return result;
}

//发送 HTTP 请求
public static boolean sendMessage(SocketChannel client, String msg) {
    try {
        ByteBuffer buf = ByteBuffer.allocate(1024);
        buf = ByteBuffer.wrap(msg.getBytes());
        client.write(buf);
    } catch (IOException e) {
        return true;
    }
    return false;
}
```



## 2.6 RSS 抓取

一些博客或者新闻网站提供 RSS (Really Simple Syndication) 格式的输出, 方便程序快速访问更新的信息。RSS 抓取的第一步是解析 RSS 数据源。Informa(<http://informa.sourceforge.net/>) 提供了一个解析包。ROME (<https://rome.dev.java.net/>) 是另外一个常用的解析包。WebNews Crawler (<http://senews.sourceforge.net/>) 是一个在 Informa 基础上构建的新闻爬虫。

RSS 种子 (Feed) 就是一个发布最新信息的 URL 地址。为了读取一个 RSS 种子, 首先定义读取种子的源, 然后定义构建新闻频道对象模型的 ChannelBuilder。

```
ChannelBuilder builder = new ChannelBuilder();
String url = "http://rss.news.yahoo.com/rss/topstories"; //RSS 种子
Channel channel = (Channel) FeedParser.parse(builder, url);
System.out.println("标题: " + channel.getTitle());
System.out.println("描述: " + channel.getDescription());
System.out.println("内容:");
for (Object x : channel.getItems()) { //遍历最新的新闻条目
    Item anItem = (Item) x;
    System.out.print("标题:"+anItem.getTitle() + " 描述: ");
    System.out.println(anItem.getDescription());
}
```

如何从网页中发现 RSS 种子? 下面是一个对 RSS 种子的描述:

```
<link href="http://blog.csdn.net/myth1979/rss.aspx" title="RSS" type=
"application/rss+xml" rel="alternate" />
```

根据 type= “application/rss+xml”可以确定 link 标签中包含的 URI <http://blog.csdn.net/myth1979/rss.aspx> 是 RSS 种子。利用 HTMLParser 来解析出有效种子的程序如下所示:

```
TagNode tagNode = (TagNode)node;
String name = ((TagNode)node).getTagName();
if (name.equals("LINK") && !tagNode.isEndTag() ) {
    String href=tagNode.getAttribute("HREF");
    if(href!=null &&
        (href.indexOf("rss")>=0 ||
         href.indexOf("feed")>=0 ||
         href.indexOf("rdf")>=0||
         href.indexOf("xml")>=0||
```



```
        href.indexOf("atom")>=0))    {
//验证 Feed 的有效性
boolean ret = rParser.valid(href);
if(ret)
    rrsUrls.add(href);
if("alternate".equals(tagNode.getAttribute("REL")))
{
    outURLs.clear();
    System.out.println("end find");
    return rrsUrls;
}
}
}
if( name.equals("A") ) {
    String href = tagNode.getAttribute("HREF");
    if(href != null &&
        (href.indexOf("rss")>=0 ||
         href.indexOf("feed")>=0 ||
         href.indexOf("rdf")>=0 ||
         href.indexOf("xml")>=0 ||
         href.indexOf("atom")>=0))    {
        boolean ret = rParser.valid(href);
        if(ret)
            rrsUrls.add(href);
    }
    if(href != null &&
        outURLs!=null &&
        href.startsWith("http://") &&
        href.indexOf(domain)>=0)    {
        outURLs.add(href);
    }
}
}
```

从一个网页中发现种子的步骤说明如下。

- ① 用一个函数来验证种子是否有效。
- ② 如果这个 URI 已经指向一个种子，则只是返回它，否则分析这个页面。
- ③ 看这个页面的头信息是否包含 LINK 标签。
- ④ <A>链接到同一个服务器上以“.rss”、“.rdf”、“.xml”或“.atom”结尾的种子。
- ⑤ <A>链接到同一个服务器上包含“.rss”、“.rdf”、“.xml”或“.atom”的种子。

- ⑥ <A>链接到以“.rss”、“.rdf”、“.xml”或“.atom”结尾的外部服务器种子。
- ⑦ <A>链接到包含“.rss”、“.rdf”、“.xml”或“.atom”的外部服务器种子。
- ⑧ 尝试猜测一些可能有种子的通用的地方，例如 index.xml、atom.xml 等。

RSS 抓取流程是：首先从网站中自动发现 RSS，然后遍历每一个 RSS，必要的时候分析详细页面。



## 2.7 抓取 FTP

Commons VFS (<http://commons.apache.org/vfs/index.html>) 提供一个统一的 API 用于处理各种不同的文件系统，包括 FTP 中的文件或者 Zip 压缩包中的文件。使用 VFS 遍历 FTP 中文件的例子如下所示：

```
FileSystemManager manager = VFS.getManager();
FileObject ftpFile = manager.resolveFile("ftp://hcl:hcl@localhost:21/loveapple");
FileObject[] children = ftpFile.getChildren();
System.out.println("Children of " + ftpFile.getName().getURI());
for (FileObject child : children) {
    String baseName = child.getName().getBaseName();
    System.out.println("文件名: " + baseName + " -- "
        + new String(baseName.getBytes("iso-8859-1"), "UTF-8"));
}
```



## 2.8 下载图片

为了减少存储空间，需要把图片转换成 JPG 格式。

```
public class ReadImage {
    public void download(String imageUrl, String imageName) throws Exception {
        URL url = new URL(imageUrl);

        Image src = javax.imageio.ImageIO.read(url); //构造 Image 对象
        int width = src.getWidth(null); //得到源图宽
        int height = src.getHeight(null); //得到源图长
```





```
        BufferedImage tag = new BufferedImage(width / 1, height / 1,
                                                BufferedImage.TYPE_INT_RGB);
        //绘制缩小后的图
        tag.getGraphics().drawImage(src, 0, 0, width / 1, height / 1, null);
        FileOutputStream out = new FileOutputStream(imageName);
        //输出到文件流
        JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(out);
        encoder.encode(tag); //JPEG 编码
        out.close();
    }
    public static void main(String[] args) throws Exception {
        ReadImage readImage = new ReadImage();
        readImage.download("http://www.51766.com/img/jhzdd/116704025
1883.bmp", "D:/PathSight/HH.jpg");
    }
}
```



## 2.9 图像的 OCR 识别

在抓取过程中，如果碰到包含价格或联系方式等信息的图片，需要转换成文字信息。例如这个以图片表示的价格：¥1119.00。在搜索引擎中实现图片或视频搜索时，也可以把图片或视频中的文字信息提取出来，然后再按文字来查找。

OCR（Optical Character Recognition）识别图像的核心是对图像切割和分类。图像识别过程说明如下。

- ① 对要识别的图像区域进行二值化处理；识别出字体和背景的颜色。如果有必要，还需要对图像中的干扰去噪。
- ② 对二值化处理后的图像切分；图像切分就好像裁缝裁布一样，一般从空白处切开图片。因为字号变化导致对应的字符图像尺寸相差可能达到数十倍，所以还需要把切割后的图像大小归一化。
- ③ 判断包含单个字符的图像应该识别成哪个字符，这是个分类的问题，可以采用机器学习的方法实现，是有监督的学习（supervised learning）过程。先要准备好包含对应字符的图像样本及应该识别出的字符答案作为训练库。事先训练出模型后，识别时加载用分类的方法学习出的分类模型；分类方法可以选择 SVM 或者最大熵分类等。在 OCR 识别过程中，

需要对从图像提取出的数据集进行分类，因此合适的分类器对结果的预测起着非常重要的作用。近年来，支持向量机（SVM）作为一种基于核方法的机器学习技术，不仅有强烈的理论基础而且有极好的成功经验。这里采用开源的 SVM 软件 LIBSVM 来对图像进行分类。

④ 执行分类并输出结果。如果有必要可以根据上下文猜测最有可能的输出结果以降低识别错误。

调用图像识别模块接口的代码如下所示：

```
String imgFile = "D:/lg/ocr/sample-images/ESfjydz.png";
OCR ocr = new OCR();
ocr.preload(); //学习的过程
String text = ocr.recognize
(imgFile); //识别的过程
System.out.println("\nresult:"+text);
```

### 2.9.1 图像二值化

对要识别的图像区域进行二值化处理是很重要的步骤。比如一个字母 b 这样的图像区域变成一个由 0 和 1 组成的图像，如图 2-12 所示。

在实际应用过程中图像的二值化处理很重要，否则识别出来的可能都是 1 了，也就是说图片需要预处理成 0101 这样的黑白格式。

首先要支持读入多种常见图片文件格式，例如 JPG、GIF 等。Java 中进行图像 I/O 有以下四种方法。

- Image I/O API，支持常见图片格式，从 Java 2 version 1.4.0 开始就内置在 J2SE 了。Image I/O API 提供了加载和保存图片的方法。支持读取 GIF、JPEG 和 PNG 图像，也支持写 JPEG 和 PNG 图像，但是不支持写 GIF 文件。
- Java 2D API 提供了一些用于图像表示的类以及一些新的支持图像处理相关操作的类。这些类都包含在 java.awt 包、java.awt.image 包以及 6 个新增加的包中。这些类大大加强了 Java 的 2D 图像处理能力。因此，可以通过 Java 2D API 中图像类制作图像。Java 2D API 和 Image I/O API 的区别是：Java 2D API 偏向于图形，画点画线之类；而 Image I/O API 主要用于读写图片，偏向于图像。

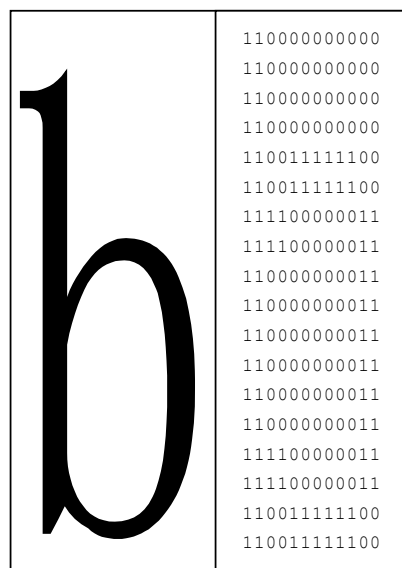


图 2-12 字符 b 及其由 0 和 1 组成的二值化结果



- Java Advanced Imaging (JAI) 中的 Image I/O Tools, 支持更多图片类型, 其介绍主页在 <https://jai-imageio.dev.java.net/>。JAI 是一个关于图像处理的框架, 很庞大, 其中仅仅 jai-imageio 是关于图像 I/O 的。
- JAI 的 `com.sun.media.jai.codec` 也有一定的图像解码能力。如果仅仅想读取常见格式的图片, 不需要用 JAI 这么高级这么庞大的东西, 用 Java Image I/O API 即可。

下面介绍 Java Image I/O API。Java Image I/O API 主要在 `javax.imageio` 包中。JDK 已经内置了常见图片格式的插件, 但它提供了插件体系结构, 第三方也可以开发插件支持其他图片格式。

`javax.imageio.ImageIO` 类提供了一组静态方法进行最简单的图像 I/O 操作。读取一个标准格式 (GIF、PNG 或 JPEG) 的图片很简单:

```
File f = new File("c:\\images\\myimage.gif");
BufferedImage bi = ImageIO.read(f);
```

Java Image I/O API 会自动探测图片的格式并调用对应的插件进行解码, 当安装了一个新插件, 新的格式会被自动理解, 程序代码不需要改变。

有些待识别的图片存在干扰, 可以考虑按颜色分辨出干扰。例如, 有的图片只有三种颜色: 白色背景、红色字、黑色水平线, 其中黑色水平线是干扰。我们需要把黑色去掉和只留下白色背景, 红色字。

将彩色图片黑白化的常规做法是先将其转化为灰度图, 再把灰度图转化为黑白图。为了用计算机来表示和处理颜色, 必须采用定量的方法来描述颜色, 即建立颜色模型。常用的有以红、绿、蓝为原色的 RGB 颜色模型和描述色彩饱和度的 HSV 颜色模型。RGB 彩色包含了亮度信息, 即 YUV 模型中的 Y 分量, 灰度计算公式如下所示:

$$\text{gray} = 0.229 * r + 0.587 * g + 0.114 * b$$

计算一个 RGB 颜色模型表示像素的灰度代码如下所示:

```
/**
 * 计算一个 RGB 颜色的灰度
 * @param pixel
 * @return 对应的灰度
 */
private static int getGray(int pixel) {
```

```

int r = (pixel >> 16) & 0xff;
int g = (pixel >> 8) & 0xff;
int b = (pixel) & 0xff;

//按公式计算出灰度值
int gray = (int) (0.229 * r + 0.587 * g + 0.114 * b);

return gray;
}

```

统计图片灰度直方图的代码如下所示：

```

int grayValues[]; //每个像素对应的灰度值

BufferedImage bi = readImageFromFile(imageFile);

//得到图片的宽和高
int width = bi.getWidth(null);
int height = bi.getHeight(null);

//读取像素
int pixels[] = new int[width * height];
bi.getRGB(0, 0, width, height, pixels, 0, width);

//计算每个像素的灰度，保存下来
grayValues = new int[width * height];
for(int i = 0; i < width * height; i++) {
    grayValues[i] = getGray(pixels[i]);
}

int hist[] = new int[256]; //存放每种灰度的出现次数

for(int i = 0; i < grayValues.length; i++) {
    hist[grayValues[i]]++; //统计每种灰度出现的次数，即得到直方图
}

```

可以根据灰度值来确定像素的二值化结果。最简单的方式是固定阈值法。例如，若灰度大于 128，则认为是白色；灰度小于 128，则认为是黑色。

二值化的双峰法原理是：认为图像由前景和背景组成，在灰度直方图上，前后二景都形成高峰，在双峰之间的最低谷处就是图像的阈值所在，可根据此值对图像进行二值化处理。除此之外，还可以考虑对颜色聚类，用 K 平均聚类方法把颜色聚成两类。



## 2.9.2 切分图像

对字符区域定位的 `CharRange` 类把带有字符的区域从图像中分离出一个矩形框出来。

```
public class CharRange {
    int x; //横坐标位置
    int y; //纵坐标位置
    int width; //宽度
    int height; //高度
}
```

对图片进行垂直和水平方向扫描的 `Entry` 类实现切割字符并且把字符大小归一化，也就是统一图像的宽度和高度。`Entry` 的主要成员变量及方法如下所示：

```
public class Entry {
    static final int DOWNSAMPLE_WIDTH = 12; //样本数据宽度
    static final int DOWNSAMPLE_HEIGHT = 18; //样本数据高度
    protected Image entryImage; //存储检测的图像
    protected Graphics entryGraphics; //处理图形图像
    protected int pixelMap[]; //存储图像像素

    //水平扫描图像并进行像素检测
    protected boolean hLineClear(int x, int w, int y) {
        int totalWidth = entryImage.getWidth(null);
        for (int i = x; i <= w; i++) {
            if (pixelMap[(y * totalWidth) + i] != -1)
                return false;
        }
        return true;
    }

    //垂直扫描图像并进行像素检测
    protected boolean vLineClear(int x) {
        int w = entryImage.getWidth(null);
        int h = entryImage.getHeight(null);
        for (int i = 0; i < h; i++) {
            if (pixelMap[(i * w) + x] != -1)
                return false;
        }
        return true;
    }

    //找到水平扫描时的上边界和下边界
    void findVBound(CharRange cr) {
```

```

        for (int i = 0; i < cr.height; i++) {
            if (!hLineClear(cr.x, cr.width, i)) {
                cr.y = i;
                break;
            }
        }
        for (int i = cr.height - 1; i >= 0; i--) {
            if (!hLineClear(cr.x, cr.width, i)) {
                cr.height = i;
                break;
            }
        }
    }
}

//找到垂直扫描时的左边界和右边界
protected ArrayList<CharRange> findHBounds(int w, int h) {
    ArrayList<CharRange> bounds = new ArrayList<CharRange>();
    int begin = 0;
    int end = w;
    boolean lastState = false;
    boolean curState = false;
    for (int i = 0; i < w; i++) {
        if (vLineClear(i)) {
            System.out.println("find blank:" + i);
            curState = false;
        } else {
            curState = true;
        }
        if (!lastState && curState) {
            begin = i;
        } else if (lastState && !curState) {
            end = (i - 1);
            CharRange cr = new CharRange(begin, 0, end, h);
            bounds.add(cr);
        }
        lastState = curState;
    }
    if (curState) {
        CharRange cr = new CharRange(begin, 0, w - 1, h);
        bounds.add(cr);
    }
    return bounds;
}

//发现图像边界

```



```
protected ArrayList<CharRange> findBounds(int w, int h) {
    ArrayList<CharRange> bounds = findHBounds(w, h);
    for (CharRange cr : bounds) {
        findVBound(cr);
    }
    return bounds;
}

//对样本数据进行采样、归一化
public ArrayList<SampleData> downSample() {
    int w = entryImage.getWidth(null);
    int h = entryImage.getHeight(null);
    ArrayList<SampleData> samples = new ArrayList<SampleData>();
    PixelGrabber grabber = new PixelGrabber(entryImage, 0, 0, w, h, true);
    grabber.grabPixels();
    pixelMap = (int[]) grabber.getPixels();
    ArrayList<CharRange> bounds = findBounds(w, h);
    for (CharRange cr : bounds) {
        SampleData data = new SampleData("?", DOWNSAMPLE_WIDTH,
            DOWNSAMPLE_HEIGHT);
        System.out.println(cr);
        double ratioX = (double) (cr.width - cr.x + 1)
            / (double) DOWNSAMPLE_WIDTH;
        double ratioY = (double) (cr.height - cr.y + 1)
            / (double) DOWNSAMPLE_HEIGHT;
        for (int y = 0; y < data.getHeight(); y++) {
            for (int x = 0; x < data.getWidth(); x++) {
                if (downSampleQuadrant(x, y, ratioX,
                    ratioY, cr.x, cr.y))
                    data.setData(x, y, true);
                else
                    data.setData(x, y, false);
            }
        }
        data.ratio = (double) (cr.width - cr.x + 1)
            / (double) (cr.height - cr.y + 1);
        data.ratio = (data.ratio - 1) / 4;
        samples.add(data);
    }
    return samples;
}

//在样本数据的指定范围内进行像素扫描
protected boolean downSampleQuadrant(double x, double y, double ratioX,
    double ratioY, double downSampleLeft, double downSampleTop) {
```

```

int w = entryImage.getWidth(null);
int startX = (int) (downSampleLeft + (x * ratioX));
int startY = (int) (downSampleTop + (y * ratioY));
int endX = (int) (startX + ratioX);
int endY = (int) (startY + ratioY);
for (int yy = startY; yy <= endY; yy++) {
    for (int xx = startX; xx <= endX; xx++) {
        int loc = xx + (yy * w);
        if (pixelMap[loc] != -1)
            return true;
    }
}
return false;
}
}

```

### 2.9.3 SVM 分类

为了调用通用的模式识别代码来识别采样后的字符图像代表哪个字符，需要把这个图像识别问题抽象为一般的分类问题。对于一个输入对象  $x$ ，有对应的输出类型  $y$ 。在这里，考虑对数字图片分类，输入是采样后的“0101”序列，输出则是 0~9 十个数字。“0101”序列中的每一位作为一个特征。训练集中的每个  $x_i$  都有对应的  $y_i$  这样的  $m$  个训练实例，记作： $(x_1, y_1); \dots; (x_m, y_m)$ 。其中每个  $x_i$  有  $k$  个特征。

SVM 的分类方法可以分为训练过程和识别过程：在训练过程中，要准备一些识别好的图片，学习出分类模型；在识别过程中，调用学习好的分类模型来分类。LIBSVM (<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>) 是台湾大学的林智仁教授开发的可以解决分类问题的 SVM 软件包。LIBSVM 支持多种编程语言，这里采用 Java 版本。LIBSVM 使用 `svm_train` 方法训练并返回一个支持向量机模型，然后可以调用 `svm_predict` 方法根据已训练好的模型进行预测。LIBSVM 的输入特征是整数，输出特征是正整数表示的类型。在 `svm_node` 类的实例中用稀疏的方式存储单个训练向量中的单个特征。

```

public class svm_node{
    public int index; //特征编号
    public double value; //特征对应的值
}

```

因为汉字是大字符集，识别比较困难，一般以笔画为依据。对于价格和邮件地址等单纯由数字和字母组成的文本，可以简单地采用像素取样的方法。





在 `svm_problem` 类的实例中存储训练集中的所有样本及其所属类别。

```
public class svm_problem {
    //训练数据的实例数
    public int l;
    //存放它们的目标值，类型值用整型数据，回归值用实数
    public double[] y;
    //训练向量用二维矩阵表示
    //矩阵的第一个维度的长度是训练的实例数，第二个维度是特征数量
    public libsvm.svm_node[][] x;
}
```

训练数据的输入样本组成了一个 `SampleData` 实例的数组：

```
public class SampleData{
    protected boolean grid[][];//用二维布尔数组存储的采样数据
    public double ratio; //宽高比，即 width/height
    protected String letter; //对应的字符或字符串
}
```

学习输入样本，得到分类模型：

```
//输入特征数量
int featureNum = Entry.DOWNSAMPLE_HEIGHT * Entry.DOWNSAMPLE_WIDTH + 1;
int outputNumber = sampleList.size();

//准备好训练集
TrainingSet set = new TrainingSet(featureNum, outputNumber);
set.setTrainingSetCount(sampleList.size());

for (int t = 0; t < sampleList.size(); t++) {
    int idx = 0;
    SampleData ds = (SampleData) sampleList.get(t);
    for (int y = 0; y < ds.getHeight(); y++) {
        for (int x = 0; x < ds.getWidth(); x++) {
            set.setInput(t, idx++, ds.getData(x, y) ? .5 : -.5);
        }
    }
    set.setInput(t, idx++, (ds.ratio));
    set.setOutput(t, t);
}

svm_parameter param = new svm_parameter();//设置模型参数
param.svm_type = svm_parameter.C_SVC;//标准算法
param.kernel_type = svm_parameter.RBF; //训练采用 RBF 核函数
```

```

param.degree = 3; //多项式核函数的阶次
param.gamma = 1.0 / featureNum;
param.coef0 = 0;
param.nu = 0.5;
param.cache_size = 100; //缓存内存大小设置为 100MB
param.C = 1;
param.eps = 1e-3;
param.p = 0.1;
param.shrinking = 1; //使用启发式
param.probability = 0;
param.nr_weight = 0;
param.weight_label = new int[0];
param.weight = new double[0];

svm_problem prob = new svm_problem();//训练集

prob.l = outputNumber;
prob.x = new svm_node[prob.l][];
prob.y = new double[prob.l];
for (int i = 0; i < prob.l; i++){//设置输入及对应的输出
    prob.x[i] = set.getInputSet(i);
    prob.y[i] = set.getOutput(i);
}

//检查参数的有效性
String error_msg = svm.svm_check_parameter(prob, param);

//如果有错误则退出
if (error_msg != null) {
    System.err.print("Error: " + error_msg + "\n");
    System.exit(1);
}

//执行训练
model = svm.svm_train(prob, param);

```

执行对指定图片的分类过程:

```

LoadImage li = new LoadImage();//加载图像

Entry entry = new Entry();
entry.entryImage = li.loadImageFromFile(new File(imgFile));//加载图像

ArrayList<SampleData> samples = entry.downSample();//采样

```



```
//特征数组
svm_node[] input =
    new svm_node[Entry.DOWNSAMPLE_WIDTH* Entry.DOWNSAMPLE_HEIGHT + 1];

StringBuilder text = new StringBuilder();//存放识别结果

for (SampleData ds : samples) { //分别识别每个切分出的图像
    int idx = 0;
    for (int y = 0; y < ds.getHeight(); y++) {
        for (int x = 0; x < ds.getWidth(); x++) {
            input[idx] = new svm_node();//设置分类特征，把采样点作为分类特征
            input[idx].index = idx + 1;
            input[idx].value = ds.getData(x, y) ? .5 : -.5;//设置分类特征值
            ++idx;
        }
    }

    input[idx] = new svm_node();
    input[idx].index = idx + 1;
    input[idx].value = ds.ratio;

    int best = (int) svm.svm_predict(model, input);//用 SVM 方法分类

    String result = map[best];//取得分类 Id 对应的字符串

    //对全黑图像字符的特殊处理
    if (result.equals("1") || result.equals(".") || result.equals("-")) {
        if (ds.ratio == 0) {
            result = ".";
        } else if (ds.ratio < 0) {
            result = "1";
        } else if (ds.ratio > 0) {
            result = "-";
        }
    }

    text.append(result);
}
```

尽管 SVM 分类方法返回的结果准确度已经很高，但是仍然可能把字母“o”识别成数字“0”，在这种图像过于相似的情况下要靠上下文来识别，比如后面是 h，则前面的符号更可能是字母“o”而不是数字“0”，实现上可以参考 HMM（隐马模型）。



## 2.10 Web 结构挖掘

有一些垃圾网页，虽然包含大量的查询词，但却并不是用户需要的文档。网页本身的重要性在网页排序中起着很重要的作用。PageRank 和 HITs 等算法可以根据 Web 结构自动学习出网页的重要性。图 2-13 是一个根据超链接生成的 Web 结构图，其中每个节点代表一个网页。

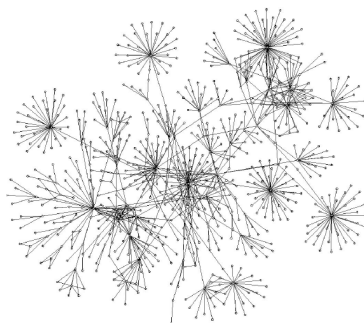


图 2-13 Web 结构图

### 2.10.1 存储 Web 图

挖掘网页之间的相互引用关系可以用来提高搜索结果排序或分析相似网页等。互联网搜索引擎需要连接服务器，能支持在网络图上快速查询连接。典型的连接查询是哪些 URL 链接到一个指定的 URL。为此，希望在内存中存储 URL 外指链接和指向该 URL 链接的映射。连接查询的应用包括爬虫控制、网络图分析、复杂的抓取优化和链接分析。

可以把每个网页看成一个节点，网页间的链接关系是有向边。几百万以上的网页将会抽象成一个包含几百万以上节点的图，因此不能简单地使用邻接表或矩阵完全在内存中处理这样大的图，往往需要通过压缩文件来存储 Web 图。

因为 BerkeleyDB 可以用来快速高效地存储海量数据。先采用它实现一个简单的 Web 图。来源 URL 和目的 URL 可以定义成一个多对多的映射。

```
@Entity
public class Link {
    @PrimaryKey public String fromURL;
    @SecondaryKey(related=Relationship.MANY_TO_MANY)
    public HashSet<String> toURL = new HashSet<String>();
}
```

在 WebGraph 类中定义从源 URL 到目标 URL 的主索引和目标 URL 到源 URL 的二级索引。

```
private PrimaryIndex<String,Link> outLinkIndex;
private SecondaryIndex<String,String,Link> inLinkIndex;
```

记录链接到 WebGraph。



```
public void addLink (String fromLink, String toLink) throws DatabaseException{
    Link outLinks = new Link();
    outLinks.fromURL = fromLink;
    outLinks.toURL = new HashSet<String>();
    outLinks.toURL.add(toLink);

    boolean inserted = outLinkIndex.putNoOverwrite(outLinks);

    if(!inserted){
        outLinks = outLinkIndex.get(fromLink);
        outLinks.toURL.add(toLink);
        outLinkIndex.put(outLinks);
    }
}
```

最后实现功能：取得指向一个 URL 地址的 Link 列表。

```
public String[] inLinks ( String URL ) throws DatabaseException {
    EntityIndex<String,Link> subIndex = inLinkIndex.subIndex(URL);
    String[] linkList = new String[(int)subIndex.count()];
    int i=0;
    EntityCursor<Link> cursor = subIndex.entities();
    try {
        for (Link entity : cursor) {
            linkList[i++] = entity.fromURL;
        }
    } finally {
        cursor.close();
    }
    return linkList;
}
```

假设有 40 亿个网页，每个网页有 10 个链接到其他页面。需要 32 位（4 个字节）来指定每个链接的源和目标，要求内存的总字节数  $4 \times 10^9 \times 10 \times 8 = 3.2 \times 10^{11}$ 。因为每个链接涉及源和目标两个网址，所以最后乘以 8，而不是 4。

可以利用 Web 图的一些基本性质使内存的使用要求降低到 10%。初看起来，这似乎是一个数据压缩的问题，对于这个问题有一些标准的解决方案。然而，我们的目标不是简单地压缩网络图以便内存能放下，还需要有效地支持连接查询。索引压缩和 Web 图的压缩问题类似。

假设每个网页用一个唯一的整数表示，最简单的存储方法是如果有网页  $P_1 \rightarrow P_2$ ，则用一行“ $P_1 P_2$ ”表示。还有一种方法是对 Web 图建立一个邻接表，类似反向索引：每行对应一个页面，行之间用网页对应的整数排序。任何网页  $p$  的对应行包含一个排好序的整数数

组，每一个整数表示对应的网页链接到  $p$ 。这个表方便查询哪些网页链接到  $p$ 。用类似的方式，还可以建立由包含  $p$  指向页面的条目组成的表。这种邻接表的表现形式所占空间比最简单的表现形式减少一半。

下面的描述将集中讨论从每一个页面发出的链接表，把这些技术应用到指向每个网页的链接也是一样的。为进一步减少表的存储空间，我们利用如下几点。

- 数组相似性：因为表中许多行的某些项是相同的，所以如果我们为几个相似行用一个原型行来代表，那么剩下的差异就可以根据原型行简洁地表现出来。
- 局域性：从一个网页发出的许多链接都是到邻近的网页。例如，在同一台主机上的网页，因此建议在编码目标连接的时候，使用小的整数来节约空间。可以使用差分编码的排序数组来实现这一点。不是直接存储每一条链接的终点网页的编号，而是存储与前一项的偏移量。

现在来实现这些压缩技术。先把每个 URL 作为一个字符串并对这些字符串排序。图 2-14 显示了这些排好序了的表的一部分。对于一个网页字典排序，URL 的域名部分应该被反转，因此 `ww.lietu.com` 变为 `com.lietu.www`，但是如果主要关心的是和本地一台主机上的链接有关，就没有必要这样做。

对每个 URL，取得它在排序后的 URL 地址列表中的位置作为表示它的唯一整数。图 2-15 显示这样编号后的 Web 图的例子。在这个例子的序列中 `www.lietu.com/demo` 分配给整数 2，因为它在数组中排第二。

1:	www.lietu.com
2:	www.lietu.com/demo
3:	www.lietu.com/english
4:	www.lietu.com/news
5:	www.lietu.com/job
6:	www.lietu.com/train

图 2-14 URL 字典排序

1:	1,2,4,8,16,32,64
2:	1,4,9,16,25,36,49,64
3:	1,2,3,5,8,13,21,34,55,89,144
4:	1,4,8,16,25,36,49,64

图 2-15 链接表的四行片断

下面来分析 Web 图的相似性和局域性。许多网站都有模板，站点的每个网页链接到一个固定的网页集合，固定的网页集合常常包括版权说明页、网站首页、站点地图页等。在这种情况下，Web 图中相应网页的行有许多项是相同的。此外，按照 URL 字典的排序，从一个网站出来的网页极有可能作为 Web 图中临近的行。



我们采取引用压缩（Reference Compression）策略：从上往下遍历每一行，根据前面 7 行来编码当前行。在图 2-9 的例子中，可以编码第 4 行和偏移量为 2 的行相同，只需要用 8 代替 9 即可。仅使用前面 7 行有两个有利条件：

- 仅用 3 位来表示偏移量；这种选择在概率上是最优化的。
- 固定的最大偏移量为一个像 7 这样小的值，避免了在许多可能表示当前行的候选原型中选择，因为这是代价昂贵的搜索。

有时候前面 7 行没有一个能很好表达当前行的原型，例如在每个不同站点的边界。这时，简单地表示该行作为空行开始并向其中添加每一个整数。Web 图的每行中，使用差分编码（Gap Encoding）来存储增量，而不是实际值，基于值的分布来编码这些增量，可以减少更多的空间。这里提到的一系列技术使平均每个链接所需空间减少到 3 位，而不是最简单方法的 64 位。

虽然这些方法能够把 Web 图的表示存入内存中，但是仍然需要支持连通性查询的要求。如何查询一个网页指向哪些网页？首先，我们需要从哈希表中查询 URL 对应的行号。接下来，需要跟踪偏移量指示的其他行来重建这些被压缩了的项，而且，这个间接过程可能重复多次。

然而，在实践中不会经常发生这样的事情。Web 图的构建过程中可以引入启发式控制：检查前面 7 行哪个作为当前行的原型时，当前行和候选原型之间有个相似性的阈值。必须谨慎地选择阈值，如果设置得太高，会很少使用原型并重新表示许多行。如果设置得太低，许多行将会用原型项表示，在查询时的重建速度就会变慢。

WebGraph (<http://webgraph.dsi.unimi.it/>) 是一个开源项目，它采用了上面介绍的引用压缩技术把图压缩成 BV 格式。

因为 WebGraph 存储的节点是整型，不是直接的 URL 地址，所以首先需要把 Web 网页映射成 0 到  $n$  的节点编号。然后形成有向边的图，例如 example.arcs 文件中包括 (0,1,2,3,4) 5 个节点组成的图，这个文件的内容如下：

```
0 2
1 2
1 4
2 3
3 4
```

下面这个命令将会产生一个压缩图到 `bvexample` 文件：

```
#java it.unimi.dsi.webgraph.BVGraph -g ArcListASCIIGraph example.arcs
bvexample
```

生成压缩图以后我们可以统计图的出度：

```
#java it.unimi.dsi.webgraph.examples.OutdegreeStats -g BVGraph bvexample
```

执行结果如下所示：

```
The minimum outdegree is 0, attained by node 4
The maximum outdegree is 2, attained by node 1
The average outdegree is 1.0
```

### 2.10.2 PageRank 算法

为了得到更好的搜索结果，使搜索引擎自动抵制那些堆砌关键词的垃圾网页，需要计算网页本身的重要性。假设用户在随机访问互联网的每个页面。看完一个页面后，可能再通过这个页面的链出链接访问其他的网页。为了在搜索结果中更好地对网页排序，考虑把用户更有可能访问的页面排在前面。有的网页被链接的次数多，所以用户访问的可能性更大。

一个简单的方法是用一个网页的入链数量表示此网页的重要程度。对应的概念叫做链接人气值。一般来说，如果从其他网页链接到一个网页的数量越多，那么这个网页就越重要。链接人气值的概念通常可以避免那些只被创造出来欺骗搜索引擎并且没有任何实际意义的网页得到好的等级，然而，许多网站管理员为了一个网页取得更好的排名，他们从大量其他没有意义的网页链接到该网页。链接人气值的问题在于没有考虑入站链接本身的权威性。

与链接人气值相比较，**PageRank** 的概念并不是简单地根据入站链接的总数评价一个网页的重要度。**PageRank** 对入站链接做加权计算，越是重要的网页通过链接指向一个网页，则这个网页就越重要。

首先介绍简化版本的 **PageRank** 计算方法。假设有  $n$  个网页指向网页 **A**，这  $n$  个网页分别是  $T_1$ 、... $T_i$ ，...  $T_n$ 。则网页 **A** 的 **PageRank** 值计算方法是：

$$PR(A) = PR(T_1)/C(T_1) + \cdots + PR(T_n)/C(T_n)$$

这里， $PR(T_i)$  是链接到网页 **A** 的网页  $T_i$  的 **PageRank** 值； $C(T_i)$  表示网页  $T_i$  的出度链接



数量。可以把链接比做互联网中的钞票，如果某个网站的钞票发行太多，那么它的钞票就要贬值，所以这里要除以  $C(T_i)$ 。

例如图 2-16 中第一个节点的 PageRank 值计算如下：

$$\begin{aligned}
 &= (\text{ID}=2 \text{ 发出的 Rank}) + (\text{ID}=3 \text{ 发出的 Rank}) + (\text{ID}=5 \text{ 发出的 Rank}) + (\text{ID}=6 \text{ 发出的 Rank}) \\
 &= 0.166 + 0.141/2 + 0.179/4 + 0.045/2 \\
 &= 0.30375
 \end{aligned}$$

下面我们用矩阵形式表示 PageRank 的计算过程。Google 矩阵是一个随机矩阵，用来表示一个图，图中的边表示网页之间的连接。随机矩阵的特点是每行值的和是 1。

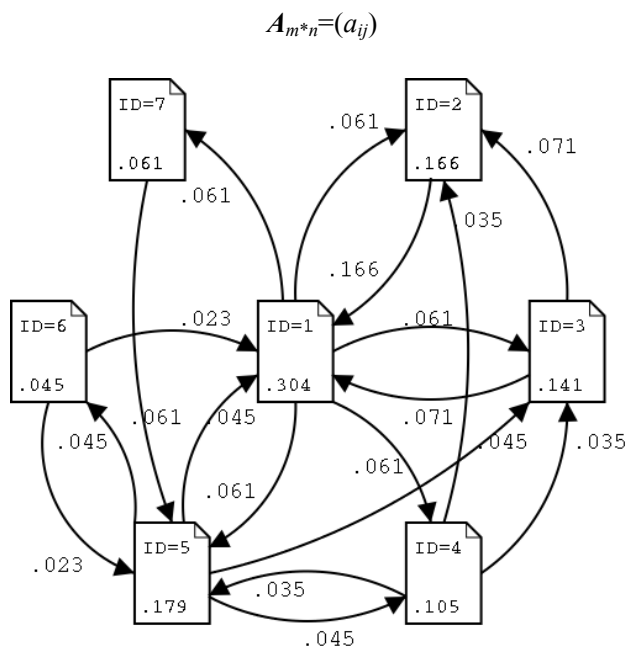


图 2-16 Web 图

矩阵中元素  $a_{ij}$  的取值方法是：如果存在从网页  $i$  指向网页  $j$  的超链接，则  $a_{ij}=1/N(i)$ ，这里  $N(i)$  表示从网页  $i$  向外的链接数目。如果不存在这样的链接，则  $a_{ij}=0$ 。例如图 2-10 表示的 Web 图的 Google 矩阵是：

$$\begin{pmatrix} 0 & 1/5 & 1/5 & 1/5 & 1/5 & 0 & 1/5 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 1/3 & 0 & 1/3 & 0 & 0 \\ 1/4 & 0 & 1/4 & 1/4 & 0 & 1/4 & 0 \\ 1/2 & 0 & 0 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

令  $x = \text{PageRank}$ ,  $M = A^T$ , 则可以通过迭代的方法计算:

$$x = M \times x$$

用 Google 矩阵计算 PageRank 的实现代码如下所示:

```
//p 是 A 矩阵的转置
double[][] p = Matrix.transpose(counts);

//初始化 pagerank 向量
double[] rank = new double[N];
for (int i = 0; i < N; i++) {
    rank[i] = 1.0/N; //N 是网页的总数, 给所有的页面 rank 赋初始值 1/N
}

int T=100;
//执行乘积的 T 次迭代
for (int t = 0; t < T; t++) {
    rank = Matrix.multiply(p, rank);
}
```

标准的 PageRank 算法是:

$$\text{PR}(A) = (1-d)/N + d (\text{PR}(T_1)/C(T_1) + \dots + \text{PR}(T_n)/C(T_n))$$

这里,  $N$  表示网页总数;  $\text{PR}(A)$  表示网页  $A$  的 PageRank 值;  $\text{PR}(T_i)$  表示链接到  $A$  的网页  $T_i$  的 PageRank 值;  $C(T_i)$  表示网页  $T_i$  的出站链接数量;  $d$  是阻尼系数, 取值范围是:  $0 < d < 1$ 。

首先, PageRank 并不是将整个网站排等级, 而是以单个页面计算的。其次, 页面  $A$  的 PageRank 值取决于那些连接到  $A$  的页面的 PageRank 的递归值。

每个  $PR(T_i)$  值并不是同等程度影响  $PR(A)$ 。在 PageRank 的计算公式里,  $T$  对于  $A$  的影响还受  $T$  的出站链接数  $C(T)$  的影响。这就是说,  $T$  的出站链接越多,  $A$  受  $T$  的这个链接的影响就越少。

$PR(A)$  是所有  $PR(T_i)$  之和。所以, 对于  $A$  来说, 每增加一个入站链接都会增加  $PR(A)$ 。

最后, 所有  $PR(T_i)$  之和乘以一个阻尼系数  $d$ , 它的值在 0 到 1 之间。因此, 阻尼系数的使用, 减少了其他页面对当前页面  $A$  的重要度贡献。

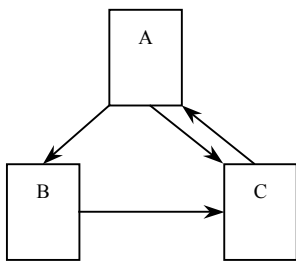


图 2-17 一个简单的 Web 图

PageRank 的特性可以通过图 2-17 表示。

假设一个小网站由三个页面  $A$ 、 $B$ 、 $C$  组成,  $A$  连接到  $B$  和  $C$ ,  $B$  连接到  $C$ ,  $C$  连接到  $A$ 。虽然 Google 实际上将阻尼系数  $d$  设为 0.85, 但这里我们为了简便计算就将其设为 0.5。尽管阻尼系数  $d$  的精确值无疑是影响到 PageRank 值的, 可是它并不影响 PageRank 计算的原理。因此, 我们得到以下计算 PageRank 值的方程:

$$PR(A) = 0.5 + 0.5 PR(C)$$

$$PR(B) = 0.5 + 0.5 (PR(A) / 2)$$

$$PR(C) = 0.5 + 0.5 (PR(A) / 2 + PR(B))$$

求解这些方程, 得到每个页面的 PageRank 值为:

$$PR(A) = 14/13 = 1.07692308$$

$$PR(B) = 10/13 = 0.76923077$$

$$PR(C) = 15/13 = 1.15384615$$

很明显所有页面 PageRank 之和为 3, 等于网页的总数。就像上面所提到的, 此结果对于这个简单的范例来说并不特殊。

对于这个只有三个页面的简单范例来说, 通过方程组很容易求得 PageRank 值。但实际上, 互联网包含数以亿计的文档, 我们是不可能解方程组的。

由于实际的互联网网页数量巨大, Google 搜索引擎使用了一个近似的、迭代的计算方法计算 PageRank 值。就是说先给每个网页一个初始值, 然后利用上面的公式, 循环进行有限次运算得到近似的 PageRank 值。我们再次使用“三页面”的范例来说明迭代计算, 这里

设每个页面的初始值为 1。计算过程如表 2-8 所示。

表 2-8 PageRank 迭代计算方法

迭代次数	PR(A)	PR(B)	PR(C)
0	1	1	1
1	1	0.75	1.125
2	1.0625	0.765625	1.1484375
3	1.07421875	0.76855469	1.15283203
4	1.07641602	0.76910400	1.15365601
5	1.07682800	0.76920700	1.15381050
6	1.07690525	0.76922631	1.15383947
7	1.07691973	0.76922993	1.15384490
8	1.07692245	0.76923061	1.15384592
9	1.07692296	0.76923074	1.15384611
10	1.07692305	0.76923076	1.15384615
11	1.07692307	0.76923077	1.15384615
12	1.07692308	0.76923077	1.15384615

实际计算大的 Web 图时，往往需要上百次计算才能得到稳定的 PageRank 值。这里只给出了 12 次迭代计算作为示例。

根据标准的 PageRank 算法，Page 类的 recalculatePageRank 方法计算单个网页的 PageRank 值实现代码如下所示：

```
/**
 * 使用提供的阻尼系数重新计算这个网页的 PageRank
 *
 * @param decayFactor 模拟图遍历中的随机游走
 */
public void recalculatePageRank(double decayFactor) {

    //取得 PageRank 的收入部分：
    //每个进入网页的 PageRank 除以它的链出数之和
    double incomingPortion = 0.0;
    Iterator iterator = this.graph.getIncomingVertices(this).iterator();
    while (iterator.hasNext()) {
        RankablePage page = (RankablePage) iterator.next();
        incomingPortion += (page.getPageRank() / this.graph.outDegreeOf(
            page));
    }
}
```



```
        this.pageRank = (1.0 - decayFactor) + (decayFactor * incomingPortion);
    }
```

PageRank 类的 run 方法计算整个图中网页的 PageRank 值实现代码如下所示：

```
/**
 * 使用给定的迭代次数和阻尼系数计算图中的网页的 PageRank 值
 *
 * @param numIterations 迭代次数
 * @param decayFactor 模拟随机游走的阻尼系数
 */
public void run(int numIterations, double decayFactor) {

    //迭代式的重新计算网页的 PageRank 值
    for (int i = 1; i <= numIterations; i++) {

        //计算 PageRank 值
        Iterator pageIterator = this.graph.vertexSet().iterator();
        while (pageIterator.hasNext()) {
            RankablePage page = (RankablePage) pageIterator.next();
            if (graph.outDegreeOf(page) > 0) {
                page.recalculatePageRank(decayFactor);
            }
        }
    }

    //输出最终结果
    Iterator pageIterator = this.graph.vertexSet().iterator();
    while (pageIterator.hasNext()) {
        RankablePage page = (RankablePage) pageIterator.next();
        System.out.println(page.getUrl() + " (" + page.getPageRank() + ")");
    }
}
```

图 2-18 显示了如何在搜索引擎中使用计算出来的 PageRank 来改进搜索结果排序。在搜索系统实际运行时，因为 Web 图是动态变化的，所以网页的 PageRank 值也是动态变化的。

当把 PageRank 值大的网页排在前面的搜索引擎本身对用户访问网页的行为影响很大时，要减小 PageRank 值对于搜索结果排序的影响。因为考虑搜索引擎本身的影响后，PageRank 值大的网页访问量过高，用户会过多地访问这些网页。而引入 PageRank 只是为了让用户更快地访问到通过多次点击后会访问到的页面。

PageRank 除了可以用于评价网页的重要度，还可以评价学术论文的重要性。根据论文

间的引用关系，模仿网页间的链接关系，构建论文间的引用关系图，利用 PageRank 的思想，计算每一篇论文的重要性。

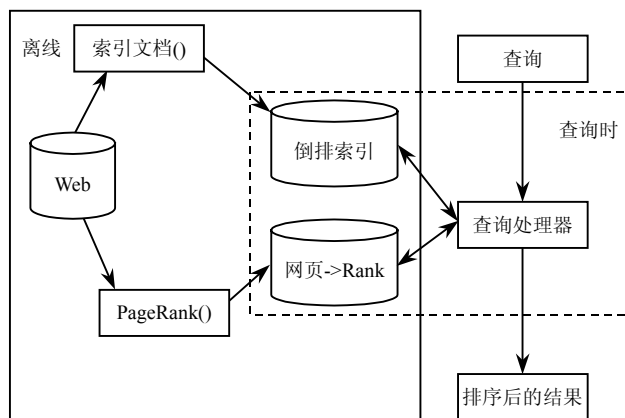


图 2-18 在搜索引擎中使用 PageRank

网络爬虫可以利用 PageRank 值决定某个 URL 所需要抓取的网页数量和深度。重要性高的网页抓取的页面数量相对多一些，反之则少一些。

可以把 PageRank 用于关键词抽取：节点是词，边是词间的共现关系。即共现的词之间，判断为互相链接，这样组成一个无向图。某个词出现的次数越多，则链向它的词可能也就越多，这样的词可能就越重要。被重要的词所链向的词也越重要。这种方法叫做 WordRank。

可以把 PageRank 算法用于句子抽取（文本摘要）：节点是句子，边是句子间的相似度。此外，还有用于文档评分的 DocRank。

### 2.10.3 HITS 算法

PageRank 算法中对于向外链接的权值贡献是平均的，也就是不考虑不同链接的重要性。而有些链接具有注释性，也有些链接是起导航或广告作用。有注释性的链接才用于权威判断。

下面介绍的 HITS 算法允许链接本身带权重。HITS 算法是由康奈尔大学的 Jon Kleinberg 博士于 1998 年首先提出的，HITS 的英文全称为 Hypertext-Induced Topic Search。

HITS 算法的实现代码如下所示：

```

public class HITS {
    /** 存储 Web 图的数据结构 */
    private WebMemGraph graph;

```



```
/** 包含每个网页的评分 */
private Map<Integer,Double> hubScores; //<id,value>

/** 包含每个网页的 Authority */
private Map<Integer,Double> authorityScores;//<id,value>

/**
 *构造函数
 */
public HITS ( WebGraph graph ) {
    this.graph = graph;
    this.hubScores = new HashMap<Integer,Double>();
    this.authorityScores = new HashMap<Integer,Double>();
    int numLinks = graph.numNodes();
    for(int i=1; i<=numLinks; i++) {
        hubScores.put(new Integer(i),new Double(1));
        authorityScores.put(new Integer(i),new Double(1));
    }
    computeHITS();
}

/**
 * 计算网页的 Hub 和 Authority 值
 */
public void computeHITS() {
    computeHITS(25);
}

/**
 *计算网页的 Hub 和 Authority 值
 */
public void computeHITS(int numIterations) {
    while(numIterations-->0 ) {
        for (int i = 1; i <= graph.numNodes(); i++) {
            Map<Integer,Double> inlinks    = graph.inLinks(new
                Integer(i));
            Map<Integer,Double> outlinks   = graph.outLinks(new
                Integer(i));
            double authorityScore = 0;
            double hubScore = 0;
            for (Integer id:inlinks.keySet()) {
                authorityScore += (hubScores.get(id)).
                    doubleValue();
            }
        }
    }
}
```

```

        for (Integer id:outlinks.keySet()) {
            hubScore += (authorityScores.get(id)).
                doubleValue();
        }

        authorityScores.put(new Integer(i),new Double
            (authorityScore));
        hubScores.put(new Integer(i),new Double(hubScore));
    }
    normalize(authorityScores);
    normalize(hubScores);
}

}

public void computeWeightedHITS(int numIterations) {
    while(numIterations-->0 ) {
        for (int i = 1; i <= graph.numNodes(); i++) {
            Map<Integer,Double> inlinks    = graph.inLinks(new
                Integer(i));
            Map<Integer,Double> outlinks   = graph.outLinks(new
                Integer(i));
            double authorityScore = 0;
            double hubScore = 0;
            for (Entry<Integer,Double> in:inlinks.entrySet()) {
                authorityScore += (hubScores.get(in.getKey())).
                    doubleValue() * in.getValue();
            }

            for (Entry<Integer,Double> out:outlinks.entrySet()){
                hubScore += (authorityScores.get(out.getKey())).
                    doubleValue() * out.getValue();
            }

            authorityScores.put(new Integer(i),new Double
                (authorityScore));
            hubScores.put(new Integer(i),new Double(hubScore));
        }
        normalize(authorityScores);
        normalize(hubScores);
    }
}

/**
 * 归一化集合
 */
private void normalize(Map<Integer,Double> scoreSet) {

```





```
        Iterator<Integer> iter = scoreSet.keySet().iterator();
        double summation = 0.0;
        while (iter.hasNext())
            summation += ((scoreSet.get((Integer) (iter.next()))).
                doubleValue());

        iter = scoreSet.keySet().iterator();
        while (iter.hasNext()) {
            Integer id = iter.next();
            scoreSet.put(id, (scoreSet.get(id)).doubleValue() /
                summation);
        }
    }

    /**
     * 返回与给定链接关联的 Hub 值
     */
    public Double hubScore(String link) {
        return hubScore(graph.URLToIdentifier(link));
    }

    /**
     * 返回与给定链接关联的 Hub 值
     */
    private Double hubScore(Integer id) {
        return (Double) (hubScores.get(id));
    }

    /**
     * 初始化与给定链接关联的 Hub 值
     */
    public void initializeHubScore(String link, double value) {
        Integer id = graph.URLToIdentifier(link);
        if(id!=null) hubScores.put(id, new Double(value));
    }

    /**
     * 初始化与给定链接关联的 Hub 值
     */
    public void initializeHubScore(Integer id, double value) {
        if(id!=null) hubScores.put(id, new Double(value));
    }

    /**
     * 返回与给定链接关联的 Authority 值
     */
```

```

public Double authorityScore(String link) {
    return authorityScore(graph.URLToIdentifier(link));
}

/**
 * 返回与给定链接关联的 Authority 值
 */
private Double authorityScore(Integer id) {
    return (Double) (authorityScores.get(id));
}

/**
 * 初始化与给定链接关联的 Authority 值
 */
public void initializeAuthorityScore(String link, double value) {
    Integer id = graph.URLToIdentifier(link);
    if(id!=null) authorityScores.put(id,new Double(value));
}

/**
 * 初始化与给定链接关联的 Authority 值
 */
public void initializeAuthorityScore(Integer id, double value) {
    if(id!=null) authorityScores.put(id,new Double(value));
}
}

```

#### 2.10.4 主题相关的 PageRank

如果不考虑出现在页面中或者指向该网页的锚点文本中的关键词,某个领域重要的页面可能 在其他领域不重要,因此产生了主题相关的 PageRank (Topic-Sensitive PageRank) 的想法。

在基本的 PageRank 算法中,使用 Web 链接结构计算出一个 PageRank 向量(向量的长度是网页的数量)来获取相对重要的网页。网页的重要性不依赖于任何特定的搜索查询词。为了达到更准确的搜索结果,可以基于一个话题集合计算一组 PageRank 向量来更精确地捕捉和一个特定主题相关的重要性概念。通过使用这些预计算的有偏见的 PageRank 向量,在查询时生成和查询相关的每个网页的重要性分值。这样可以比单一的通用 PageRank 向量生成更准确的打分。使用查询关键词的话题为满足查询词的页面计算出话题相关的 PageRank 分值。

主题相关的 PageRank 结构如图 2-19 所示,其中需要用到已经把网站分好类的目录 Yahoo!或 ODP。

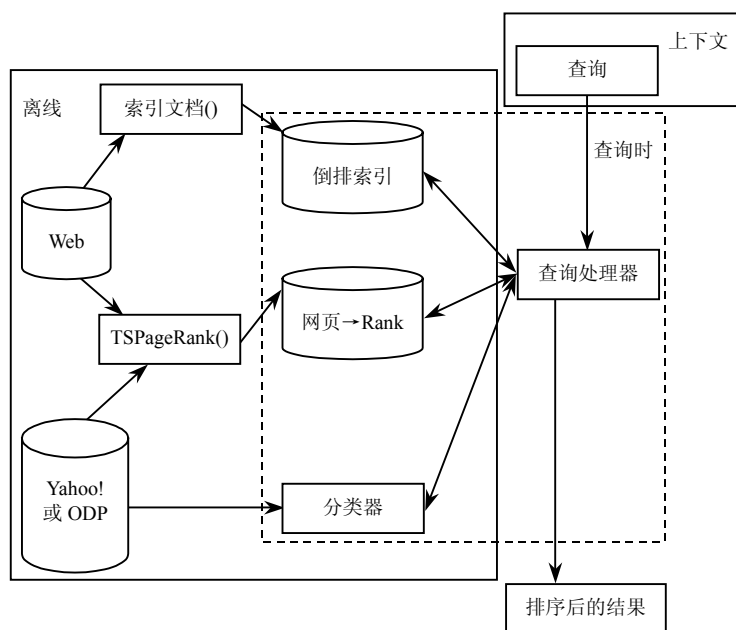


图 2-19 在搜索引擎中使用主题相关的 PageRank

改动主要在两个阶段：主题相关的 PageRank 向量集合的计算和在线查询时主题的确。首先使用一个有偏见的话题集生成一个 PageRank 向量的集合，这一步是在爬虫抓取网页时离线计算的。计算网页  $i$  在第  $j$  个类别上的 PageRank 值时，使用阻尼系数：

$$U_{ji} = \begin{cases} \frac{1}{|T_j|} & i \in T_j \\ 0 & i \notin T_j \end{cases}$$

这里的  $T_j$  是 ODP 类别  $C_j$  下的 URL 地址的集合。

在线查询时，对每个类别  $C_j$ ，计算：

$$P(C_j | q) = \frac{P(C_j)P(q | C_j)}{P(q)} \propto P(C_j) \sum P(q_j | C_j)$$

为了计算  $P(C_j)$ ，对某个用户  $k$ ，可以使用分布  $P_k(C_j)$  来反映用户  $k$  的兴趣。

最后，对于在搜索结果集中的网页  $d$  计算相关性分值：

$$\text{Score}_{qd} = \sum_j P(C_j | q) \times \text{rank}_{jd}$$

这里,  $\text{rank}_{jd}$  是网页  $d$  对于话题  $C_j$  的 PageRank 值。



## 2.11 部署爬虫

为了抓取更加稳定, 往往让爬虫运行在一个独立的控制台进程中。可以在 MANIFEST.MF 文件中声明要运行的类。通过 Ant 执行的 build.xml 来自动生成可执行的 jar 包。



## 2.12 本章小结

本章首先熟悉了网络爬虫工作的基本原理, 然后了解抓取网页的实现, 对于网页中的 JavaScript 的处理是难点问题。此外, 还介绍了 Web 图的存取和挖掘算法。接下来简单回顾一下网络爬虫的开发历史。

1994 年, 休斯敦大学的 Eichmann (<http://slis.uiowa.edu/eichmann/index.shtml>) 在美国航空航天局的资助下开发了互联网爬虫 RBSE (Repository Based Software Engineering)。RBSE 将爬虫和索引程序分离, 这样不需要重新抓取就可以更新索引。爬虫执行宽度优先遍历或者有限深度优先的遍历。Spider 程序创建和操作存储在 Oracle 数据库中的 Web 图。一个修改后的 ASCII 字符浏览器 (mite) 根据 URL 下载指定的网页存储到本地文件, 并把这个网页中的链接提取出来。

1994 年, 华盛顿大学计算机系的学生 Pinkerton 开发了分布式爬虫 WebCrawler。运行在多个机器上的爬虫代理 (Agent) 用 HTTP 下载网页并把 HTML 格式的网页解析成纯文本。WebCrawler 使用简单的广度优先遍历方法抓取部分互联网中的网页。因为早期开源数据库不稳定, 为了避免数据库服务器崩溃, 它使用商业的 Oracle 数据库作为存储核心。1997 年 Excite 收购了 WebCrawler。Pinkerton 后来担任为 Lucene 和 Solr 提供商业服务的 Lucid Imagination 公司的首席架构师。

1998 年, 斯坦福大学的学生 Brin 和 Page 用 Python 开发了分布式爬虫系统 Google Crawler。URL 服务器给几个爬虫程序发送要抓取的 URL 列表。在解析文本的时候, 把新发现的 URL 传送给 URL 服务器并检测这个 URL 是不是已经存在, 如果不存在的话, 就把该 URL 加入到 URL 服务器。爬虫程序使用了 DNS 缓存来提高 DNS 查询效率。



1999 年，Compaq 公司的 Heydon 和 Najork 开发了 Mercator。C 语言的网络 I/O 速度比 Java 快，因此直到今天很多在线运行的爬虫采用 C 或 C++ 开发，但 Mercator 却是一个用 Java 实现的成功网络爬虫。Mercator 是分布式的、模块化的。Mercator 用做 Alta Vista 搜索引擎的网络爬虫。Mercator 的模块包括可互换的“协议模块”和“处理模块”。“协议模块”负责怎样获取网页（例如使用 HTTP 协议），“处理模块”负责怎样处理页面。标准处理模块仅仅包括了解析页面和抽取新的 URL，可以用其他处理模块来索引网页中的文本，或者搜集 Web 统计数据。Mercator 使用 64 位的文档语义指纹来判断网页内容是否重复。因为布隆过滤器存在误判的情况，Mercator 没有采用 Internet Archive 爬虫中曾经采用的布隆过滤器。Mercator 用带缓存的 checksum 来判断 URL 是否抓取过。Najork 后来加入微软研究院，担任研究员。

2001 年，IBM 的 Almaden 研究中心的 Edwards 等人开发了一个与 Mercator 类似的分布式模块化的爬虫 WebFountain。它的特点是一个“控制者”机器控制一系列的“蚂蚁”机器，可以实现增量抓取。经过多次下载页面后，可以推测出每个页面的变化率，然后获得一个最大新鲜度的访问策略。商业信息网站 Factiva 使用 WebFountain 收集企业信用信息。WebFountain 是使用 C++ 实现的。

2002 年，FAST 公司的 Risvik 和 Michelsen 开发了分布式爬虫 FAST Crawler，在 Fast Search&Transfer 和 www.AllTheWeb.com 网站使用。节点之间通过 distributor 模块交换发现的链接。每个机器有一个“文档调度器”来维护一个要下载的文档队列。“文档处理器”下载完网页后把网页存储在本地存储子系统。FAST Crawler 实现了增量式抓取，优先抓取更新活跃的网页。2004 年，Yahoo 收购了 AllTheWeb 网站。Risvik 后来在 Yahoo 和 Google 工作过。

Cho 和 Garcia-Molina 在论文 Parallel Crawlers 中研究了如何设计有效的并行爬虫，在论文 Effective Page Refresh Policies For Web Crawlers 中研究了网页更新方法。

2003 年初，为了对网上的资源进行归档，建立网络数字图书馆，开发了开源网络爬虫 Heritrix。这个系统以运行时高可配置性的模块式开发，所以非常适合做实验。

早期的互联网搜索研究中有很多是关于主题爬虫的。Menczer 和 Belew 在论文 Adaptive information agents in distributed textual environments 中设想了一种为个人用户服务的、由自主软件代理的主题爬虫。用户同时输入网址以及关键字，代理就会尝试寻找对用户有用的网页，而且用户也可以对这些网页进行评估并反馈给系统。

Chakrabarti 在论文 “Focused crawling: a new approach to topic-specific web resource

discovery”中主要研究了主题爬虫，他们的爬虫使用分类来判断抓取的网页主题。他们认为对于提供主题链接方面来说，非主题爬虫无法与主题爬虫相比较。主题爬虫可以成功截获主题，广泛的网络连接结构却使非主题爬虫迅速地移动到其他主题上。

Unicode 规范是一项难以置信的复杂工作，包含了上万的字符。因为一些非西方语言特性的原因，许多象形文字都是由一组 Unicode 字符组成的。所以这个规范的细节不仅说明这些字是什么，而且解释他们是如何组合的。新的字符仍然在源源不断地加入到 Unicode。

Bergman 的论文“The Deep Web: Surfacing Hidden Value”是对深网的一个深入的研究，尽管这项研究和 Web 标准一样古老。它向我们展示了如何通过搜索引擎进行的抽样对在网络中的量指数挂钩有帮助。这个研究估算了一下大概有 5500 亿个网页存在于深网中，而与此同时却只有 10 亿个网页出现在可见网络中。He et al (2007) 提供了一个更近期的研究调查表明，深网近几年一直在持续迅速地发展中。于是一种由 Ipeirotis 与 Gravano 所提出来的叫做查询探测的用于对深网数据库进行探测分析的模型就出现了。

网站地图、robots.txt 文件、RSS feeds 和 Atom feeds 都有它们自己的规范。这些格式说明那些成功的 Web 标准一般都是很简单的。

对于一些应用程序来说，可以用数据库系统来存储爬虫下载的文件。这方面有一些教科书，例如 Garcia-Molina 的著作《Database Systems: The Complete Book》提供了许多数据库如何运行的信息，也包含一些重要功能，例如查询语句、锁、恢复等。Chang 在论文“Bigtable: 一个结构化数据的分布式存储系统”描述了 Bigtable。

另外大型互联网公司出于类似的目的：大规模分布、高吞吐量、不需要昂贵的查询语句或者详细的事务支持，纷纷建立了自己的数据库系统。DeCandia 在论文“Dynamo: Amazon’s Highly Available Key-value Store”描述的亚马逊公司的 Dynamo 系统拥有低延迟的保证。Yahoo! 使用 UDB 系统处理巨大的数据量。

DEFLATE 和 LZW 是两个文本压缩工具。DEFLATE 是现在流行的 Zip、gzip 和 zlib 这些压缩工具的基础，LZW 是 UNIX 压缩命令的基础，同样也适用于 GIF、Postscript 以及 PDF 等文件形式。Witten 写的《Managing Gigabytes: Compressing and Indexing Documents and Images》提供了一些关于文本和图片压缩算法的详细讨论。

Yu 的论文“Improving Pseudo-Relevance Feedback in Web. Information Retrieval Using Web Page Segmentation”和 Gupta 的论文“DOM-based Content Extraction of HTML Documents”是对从网页中提取正文非常有用的文献。基于内容的网页排重将在后续章节中介绍。



## 第 3 章

# 索引内容提取

搜索引擎经常要处理的文档格式包括 HTML、Word、PDF 等。这些文档格式中如 Word 和 PDF 是专有和非公开的格式，HTML 虽然是公开的标准，但是具体的实现却千差万别。而且这些文档格式往往存在不同的版本，比如 Word 包括 doc 和 docx 格式；PDF 有从 1.0 到 1.7 及其扩展版等 9 种不同的格式。



### 3.1 从 HTML 文件中提取文本

从 HTML 提取有效的文本，首先需要判断网页编码，这样才能确保不出现乱码。然后还需要从中提取需要的信息，很多网页中包括用户不想搜索的信息，例如广告、版权信息、导航条等。有很多种方法实现网页信息提取，一般可以分为两种类型：一种是针对特定的网页特征提取结构化信息；还有一种就是通用的信息提取方法，例如网页去噪。下面首先介绍字符集编码以及判断网页编码的基本过程，然后介绍网页信息提取所用到的一些工具和方法，例如 HTMLParser 和 NekoHTML。网页提取的正确率等于正确提取的文档数量除以测试集中的文档总数。

#### 3.1.1 识别网页的编码

在实现从 Web 网页提取文本之前，首先要识别网页的编码，有时候还需要进一步识别

网页所使用的语言。因为同一种编码可能对应多种语言，例如 UTF-8 编码可能对应英文或中文等任何语言。识别编码整体流程如下：

① 从 Web 服务器返回的 `content type` 头信息中提取编码，如果是 GB2312 类型的编码要当成 GBK 处理。

② 从网页的 `Meta` 标签中识别字符编码，如果和 `content type` 中的编码不一致，以 `Meta` 中声明的编码为准。

③ 如果仍然无法确定网页所使用的字符集，需要从返回流的二进制格式判断。

④ 确定网页所使用的语言，往往采用统计的方法来估计网页的语言。

从 Web 服务器返回的头信息中提取网页编码的代码如下所示。

```
final String CHARSET_STRING = "charset";
//输入头信息，返回网页编码
public static String getCharset (String content){
    int index;
    String ret = null;
    if (null != content) {
        index = content.indexOf (CHARSET_STRING);

        if (index != -1) {
            content =
                content.substring (index + CHARSET_STRING.length
                    ()).trim ();
            if (content.startsWith ("=")) {
                content = content.substring (1).trim ();
                index = content.indexOf (";");
                if (index != -1)
                    content = content.substring (0, index);

                //从字符串开始和结尾处删除双引号
                if (content.startsWith ("\"") &&
                    content.endsWith ("\"") &&
                    (1 < content.length ()))
                    content = content.substring (1, content.length () - 1);

                //删除围绕字符串的任何单引号
                if (content.startsWith ("'") &&
                    content.endsWith ("'") &&
```





```
(1 < content.length ()))
    content = content.substring (1, content.length () - 1);

    ret = findCharset (content, ret);
}
}
}

return (ret);
}
```

有的页面在头信息中不包括编码格式内容，需要从网页内部的 meta 标签中提取编码，例如下面这个：

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

下面利用 HTMLParser 包提取网页中的 meta 信息，关于 HTMLParser 将在后面的小节中详细介绍。

```
String contentCharSet = tagNode.getAttribute("CONTENT");
```

另外一个和字符编码相关的问题是，有时候碰到 GB2312 编码的网页会有乱码问题，因为浏览器能正常显示包含 GBK 字符的 GB2312 编码网页。把 org.htmlparser.lexers.InputStreamSource 中的设置编码方法修改一下，把设置字符集为 GB2312 改成 GBK：

```
public void setEncoding (String character_set){
    if(character_set!= null && character_set.toLowerCase().equals
        ("gb2312")){
        character_set = "GBK";
    }
    ...
}
```

JuniversalCharDet (<http://code.google.com/p/juniversalchardet/>) 可以根据读入的字节流自动猜测页面或文件使用的字符集。实现原理是基于统计学的字符特征分析，统计哪些字符是最常见的字符。JuniversalCharDet 可以检测的字符编码有：中文、日文、韩文、西里尔文 (Cyrillic)、希腊文、希伯来文。下面的代码调用 JuniversalCharDet 来自动判断文件的编码。

```
byte[] buf = new byte[4096];
String fileName = args[0];
java.io.FileInputStream fis = new java.io.FileInputStream(fileName);

//构建一个 org.mozilla.universalchardet.UniversalDetector 对象的实例
```

```

UniversalDetector detector = new UniversalDetector(null);

int nread;
while ((nread = fis.read(buf)) > 0 && !detector.isDone()) {
    detector.handleData(buf, 0, nread); //给编码检测器提供数据
}
//通知编码检测器数据已经结束
detector.dataEnd();

//取得检测出的编码名
String encoding = detector.getDetectedCharset();
if (encoding != null) {
    System.out.println("Detected encoding = " + encoding);
} else {
    System.out.println("No encoding detected.");
}

//在再次使用编码检测器之前,先调用 UniversalDetector.reset()
detector.reset();

```

此外还有 Jchardet 和 cpdetector。Jchardet 是基于比较老的 chardet 模块,而 JuniversalCharDet 基于新的 UniversalCharDet 模块,因此检测结果更准确。

### 3.1.2 网页编码转换为字符串编码

网页中的字符可能被转义成英文字符,例如“&#32593;&#31449;&#23548;&#33322;”是“网站导航”的转义。这类符号以“&”开始,以“;”结束。http://commons.apache.org 项目中的 StringEscapeUtils.unescapeHtml(String str) 可以把 HTML 编码的字符串转换成原文。下面的代码把输入的转义后的字符串转换成原文。

```

String htmlStr = "&#32593;&#31449;&#23548;&#33322;";
String textStr = Entities.HTML40.unescape(htmlStr);

```

### 3.1.3 使用正则表达式提取数据

图 3-1 是一个包含地址信息的网页。利用 HTMLParser 解析包可以方便地解析网页,提取想要的网页信息,但是在有些情况下利用正则表达式可能更方便(如果对正则表达式不熟悉请参考相关 Java 正则表达式教程)。同样以上述网页为例,利用正则表达式提取网页表格数据过程说明如下:

邮政编码查询					
邮编查询:	<input type="text"/>	<input type="button" value="查询"/>	区号查询:	<input type="text"/>	<input type="button" value="查询"/>
省名查询:	山西	<input type="button" value="查询"/>	市名查询:	<input type="text"/>	<input type="button" value="查询"/>
县名查询:	<input type="text"/>	<input type="button" value="查询"/>	村名查询:	<input type="text"/>	<input type="button" value="查询"/>

省名	地区	县市	乡镇村	邮政编码	区号
山西	朔州	怀仁县	鹅毛口	038301	0349
山西	朔州	怀仁县	海北头	038301	0349
山西	朔州	怀仁县	何家堡	038301	0349
山西	朔州	怀仁县	河头	038301	0349
山西	朔州	怀仁县	金沙滩	038302	0349
山西	朔州	怀仁县	里八庄	038301	0349

图 3-1 地址页面



首先，根据要提取的网页信息查看网页源代码，这里我们提取的是网页表格中地名相关的信息。查看网页源码如下所示：

```
</table> <br><table width="533" border="0" cellpadding="0" cellspacing="0" align="center" bgcolor="#3366CC"><tr align="center" bgcolor="#6699CC"><td width="70">省名</td><td width="100">地区</td><td width="100">县市</td><td width="50">乡镇村</td><td width="60">邮政编码</td><td width="50">区号</td></tr><tr align="center" onmouseout="this.style.background='#FFFFFF'" onmouseover="this.style.background='BDDFFF'" bgcolor="#FFFFFF"><td><a href=Code_Default.asp?Type=Sm&SearchKeyStr=山西>山西</a></td><td><a href=Code_Default.asp?Type=Ds&SearchKeyStr=朔州>朔州</a></td><td><a href=Code_Default.asp?Type=Xs&SearchKeyStr=怀仁县>怀仁县</a></td><td><a href=Code_Default.asp?Type=Ys&SearchKeyStr=鹅毛口>鹅毛口</a></td><td><a href=Code_Default.asp?Type=Yb&SearchKeyStr=038301>038301</a></td><td><a href=Code_Default.asp?Type=Qh&SearchKeyStr=0349>0349</a></td></tr><tr align="center">
```

根据要提取的表格信息构造正则表达式：

```
<td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td></tr>"
```

在这里要注意“(.\*?)”的用法，括号在这里表示分组，即每一个数据项表示一组，“?”表示非贪婪匹配，仅匹配到紧跟其后的“</a>”为止。

其次，根据读取的网页字符流和正则表达式提取表格信息，其相关代码如下所示：

```
//输入参数 input 是当前网页的内容字符串
public static void parserHtml(String input) {
    String regex =
        "<td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td></tr>";
    Pattern p = Pattern.compile(regex); //编译正则表达式
    Matcher m = p.matcher(input); //匹配模式
    while (m.find()) { //按行提取表格数据
        String province = m.group(1);
        String area = m.group(2);
        String city = m.group(3);
        String village = m.group(4);
        String postcode = m.group(5);
        String code = m.group(6);
        System.out.println(province); //打印出山西
    }
}
```

```
        System.out.println(area); //打印出朔州
        System.out.println(city); //打印出怀仁县
        System.out.println(villag); //打印出鹅毛口
        System.out.println(postcode); //打印出 038301
        System.out.println(code); //打印出 0349
    }
}
```

3.1.4 结构化信息提取

一般把要提取的数据结构定义成一个类，然后有一个解析网页的方法根据输入网页返回解析出来的类实例。例如，需要实现一个招聘职位的搜索引擎。从图 3-2 这个招聘网页提取职位信息。



图 3-2 招聘网页面

首先定义好用来接收网页数据的职位类：

```
public class Job {
    public String comName = null;
    public String positionName = null;
    public String email = null;
    public String releaseData = null;
    public String city = null;
    public String number = null;
    public String experience = null;
    public String salary = "面议";
    public String knowledge = null;
    public String acceptResumeLanguage = null;
    public String positionDescribe = null;
}
```



```

        public String comintro = null;
        public String comHomePage = null;
        public String comAddress = null;
        public String postCode = null;
        public String fax = null;
        public String connectPerson = null;
        public String telephone = null;
        public String languageAbility = null;
        public String funtype_big = null;
        public String funtype = null;
        public String province = null;
        public String toString(){
            return "公司名称: "+comName+"\n 职位名称: "+positionName+
                "\n 电子邮箱: " +email+"\n 发布日期: "+releaseData+
                "\n 工作地点: "+city+"\n 招聘人数: "+number+
                "\n 工作年限: "+experience+"\n 薪水范围: "+salary+
                "\n 学历: "+knowledge+
                "\n 接受简历语言: "+acceptResumeLanguage+
                "\n 职位描述: "+positionDescribe+"\n 公司简介: "+comintro+
                "\n 公司网站: "+comHomePage+"\n 地址: "+comAddress+
                "\n 邮政编码: "+postCode+"\n 传真: "+fax+
                "\n 联系人: "+connectPerson+"\n 电话: "+telephone+
                "\n 外语要求: "+languageAbility;
        }
    }

```

然后从下面这个职位发布的网页提取公司名称。查看公司名称周围的特征：

```
<td align="left" class="title02">北京亿达网通科技发展有限公司</TD>
```

可以利用 **AndFilter** 来处理：

```

//提取公司名字的 Filter
NodeFilter filter_title = new AndFilter(new TagNameFilter("TD"),
    new HasAttributeFilter("class", "title02"));

```

提取公司名称的实现代码如下所示：

```

NodeList nodelist = parser.extractAllNodesThatMatch(filter_title);
Node node_title = nodelist.elementAt(0);
String comName = extractText(node_title.toHtml());
comName = comName.replaceAll("[\t\n\f\r ]+", ""); //去掉多余的空格

```

从职位详细页面 URL 地址提取职位信息的完整过程如下所示：

```

//输入网址，返回和该网址正文信息对应的职位对象
public Job extractContent(String url){
    Job position = new Job();
    Parser parser = new Parser(url);
    //设置编码方式
    parser.setEncoding("GB2312");
    //提取公司名字的 Filter
    NodeFilter filter_title = new AndFilter(new TagNameFilter("TD"),
        new HasAttributeFilter("class", "title02"));
    //提取公司名称
    NodeList nodelist = parser.extractAllNodesThatMatch(filter_title);
    Node node_title = nodelist.elementAt(0);
    position.comName = extractText(node_title.toHtml());
    position.comName = position.comName.replaceAll("[ \t\n\f\r]
+", "");
    //提取职位名称的 Filter
    NodeFilter filter_job_name = new AndFilter(new TagNameFilter
        ("td"),
        new HasAttributeFilter("bgcolor", "#FFEEE0"));
    NodeFilter job_description_end1 = new AndFilter(new TagName
        Filter("table"),
        new HasAttributeFilter("width", "100%"));
    NodeFilter job_description_end2 = new HasAttributeFilter("cell
        padding", "5");
    NodeFilter company_description = new AndFilter(new TagName
        Filter("table"),
        new HasAttributeFilter("width", "98%"));
    //提取职位的名称
    parser.reset();
    nodelist.removeAll();
    nodelist = parser.extractAllNodesThatMatch(filter_job_name);
    Node node_job_name = nodelist.elementAt(0);
    position.positionName = extractText(node_job_name.toHtml());
    //去掉空格
    position.positionName =
        position.positionName.toString().replaceAll("[ \t\n
        \f\r] +", "");
    //提取职位描述
    NodeFilter job_description_end =
        new AndFilter(job_description_end1, job_description_end2);
    parser.reset();
    NodeList nodelist_description =
        parser.extractAllNodesThatMatch(job_description_end);
    Node node_job_description = nodelist_description.elementAt(0);
    position.positionDescribe = extractText(node_job_description.

```

```

toHtml());
position.positionDescribe =
    position.positionDescribe.replaceAll("[ \\t\\n\\f\\r ]+"," ");
//提取公司简介
parser.reset();
nodelist_description.removeAll();
nodelist_description = parser.extractAllNodesThatMatch(company_
description);
Node node_company_description = nodelist_description.elementAt(0);
//处理公司简介
position.comintro = doCompanyDescription(node_company_
description);
return position;
}

```

### 3.1.5 网页的 DOM 结构

虽然表示网页的 HTML 文档格式不如 XML 规范，但是仍然可以把它转换成 DOM（文档对象模型）树组织的结构。其中标签是 DOM 树内部的节点，而详细的文本、图像或者超链接则是叶节点。图 3-3 展示了 HTML 的一部分及其相应的 DOM 树。

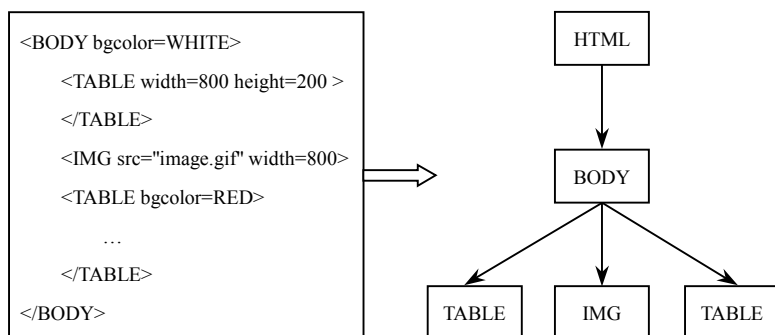


图 3-3 网页 DOM 树

在 Firefox 浏览器中，使用 DOM 查看器（DOM Inspector）和 Firebug 这两个插件工具都可以看到网页的 DOM 树。

访问 <http://www.lietu.com/>。在浏览器菜单中，选择“工具”→“DOM 查看器”命令，打开 DOM 查看器。在 DOM 查看器的左侧视图中，会看到 DOM 节点的树状图，如图 3-4 所示。

如果觉得依次展开 DOM 节点树中的每层很不方便，可以使用 Inspect Element 扩展，它能迅速找到 DOM 查看器中的指定元素。XPath Helper (<http://xpathhelper.mozdev.org/>) 可

以扩展 DOM 查看器。它在 DOM 查看器上增加了一个可以输入 XPath 的工具条，可以在当前网页计算 XPath 表达式。

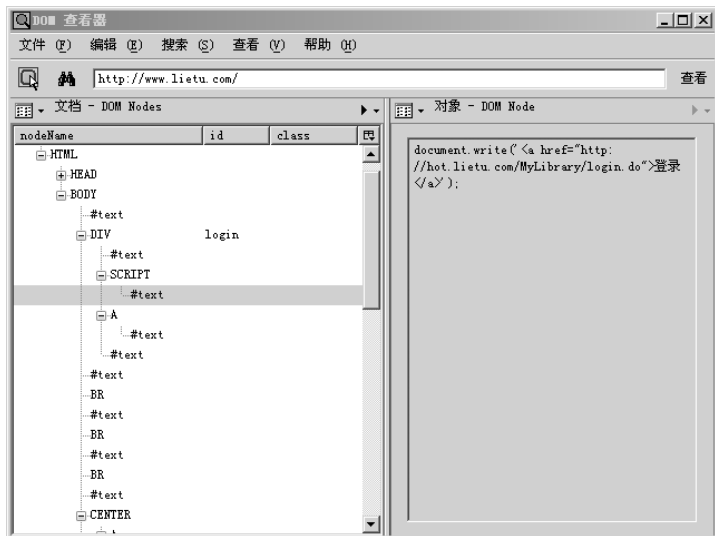


图 3-4 DOM 节点的树状图

Firebug (<http://getfirebug.com/>) 是 Firefox 的一个插件。通过 Firebug，可以在 Firefox 中查看任何页面已解析的文档对象模型 (DOM)，可以获得每个 HTML 元素、属性和文本节点的详细信息；也可以看到每个页面样式表中的所有 CSS 规则；还可以看到每个对象的所有脚本属性。在 Firefox 中安装 Firebug 插件后，输入网址“<http://www.lietu.com>”即可看到如图 3-5 所示的画面。

比较而言，DOM 查看器中显示的网页 DOM 树概念更符合下面介绍到的网页解析器 NekoHTML 所生成的 DOM 树。

### 3.1.6 使用 NekoHTML 提取信息

NekoHTML (<http://nekohtml.sourceforge.net/>) 可以解析 HTML 文档并形成标准的 DOM 树。因为 HTML 文档不如 XML 规范，可能存在一些格式不完整的元素，例如有些标签没有对应的结束标签。NekoHTML 可以通过补偿标签来整理这些有缺陷的网页，也就是说把 HTML 文档转换成 XML 格式，然后就可以通过 XML 解析器 xerces 访问 NekoHTML 解析出的网页 DOM 树。使用 NekoHTML 的 Java 项目中需要导入的包是 nekohtml.jar、xercesImpl.jar 和 xalan.jar。





图 3-5 Firebug 显示的网页 DOM 树

节点有三种类型：元素节点、注释节点和文本节点。判断各种节点类型的代码如下所示：

```
int type = node.getNodeType(); // 取得节点类型
if (type == Node.ELEMENT_NODE) {
    System.out.print("ELEMENT_NODE ");
}
else if (type == Node.COMMENT_NODE) {
    System.out.print("COMMENT_NODE ");
}
else if (type == Node.TEXT_NODE) {
    System.out.println("Text Node");
    // 打印文本节点中的文字信息
    System.out.println(node.getNodeValue());
}
```

以一个图片节点为例，Node 的基本概念如图 3-6 所示。比如要得到图片节点中 src 的值，可以调用 `node.getAttribute().getNamedItem("src")` 方法。

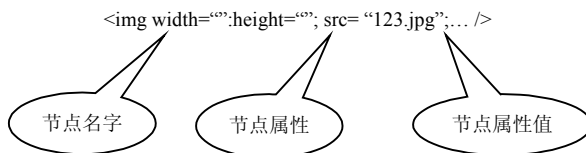


图 3-6 Node 的基本概念

还可以修改节点，比如要移除节点属性，可以用下面的方法：

```
private void removeAttribute(final Node iNode, final String iAttr) {
    iNode.getAttributes().removeNamedItem(iAttr);
}
```

每一个节点在 DOM 树中都有一个特定的位置。Node 接口中有一些发现指定节点周边节点的方法，如图 3-7 所示，具体说明如下。

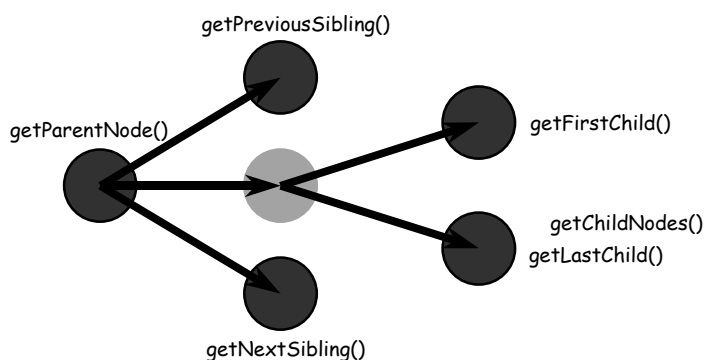


图 3-7 发现周边节点

- Node getFirstChild(): 得到第一个孩子节点。
- Node getLastChild(): 得到最后一个孩子节点。
- Node getNextSibling(): 得到下一个兄弟节点。
- Node getPreviousSibling(): 得到上一个兄弟节点。
- Node getParentNode(): 得到父亲节点。
- NodeList getChildNodes(): 得到所有的孩子节点。

下面的程序使用一个递归方法来遍历 DOM 树并打印出节点内容：

```
public static void main(String[] argv) throws Exception {
    DOMParser parser = new DOMParser(); //创建 DOM 解析器
    parser.parse("http://www.lietu.com"); //解析网页
}
```



```
        print(parser.getDocument(), ""); //从根节点开始遍历
    }

    public static void print(Node node, String indent){
        if (node.getNodeValue() != null){
            if(!"".equals(node.getNodeValue().trim())){
                System.out.print(indent);
                System.out.println(node.getNodeValue()); //打印节点内容
            }
        }
        Node child = node.getFirstChild(); //取得第一个孩子节点
        while (child != null){
            //递归调用 print 方法
            print(child, indent+" ");
            child = child.getNextSibling(); //取得孩子节点的下一个兄弟节点
        }
    }
}
```

下面的代码打印出网页中所有的链接：

```
public static void main(String[] argv) throws Exception {
    DOMParser parser = new DOMParser();
    parser.parse("http://www.lietu.com");
    Set<String> urlSet = new HashSet<String>(); //存储网页中的 Url
    extract(parser.getDocument(), urlSet);
    for (String url : urlSet) { //打印出所有的网页链接
        System.out.println(url);
    }
}

//提取链接
public static void extract(Node node, Set<String> urlSet) {
    if (node instanceof HTMLAnchorElementImpl) {
        //添加到集合中
        urlSet.add(((HTMLAnchorElementImpl)node).getAttribute("href"));
    }
    Node child = node.getFirstChild(); //取得第一个孩子节点
    while (child != null) {
        //递归调用 extract 方法
        extract(child, urlSet);
        child = child.getNextSibling(); //取得兄弟节点
    }
}
}
```

xerces-2\_9\_1 包括了一个 LSSerializer 对象可以保存 DOM 解析后的文档或节点，下面

的代码将 iNode 节点的内容输出到控制台：

```
registry = DOMImplementationRegistry.newInstance();
DOMImplementationLS impl =
    (DOMImplementationLS) registry.getDOMImplementation("LS");
LSSerializer writer = impl.createLSSerializer();
LSOutput output = impl.createLSOutput();
output.setByteStream(System.out);
output.setEncoding(System.getProperty("file.encoding"));
writer.write(iNode, output);
```

还可以使用 `writeToString` 方法将文档或文档中的节点所代表的一段网页写入字符串：

```
String nodeString=domWriter.writeToString(iNode);
```

找出符合条件的 DIV 节点下的 URL 链接。网页的 DOM 如图 3-8 所示。

```
DOMParser parser = new DOMParser();
parser.parse("http://news.steelhome.cn/z
hxx/");
Node divNode = visit(parser.getDocument());
//取得符合条件的 DIV 节点
visitDiv(divNode); //取得 DIV 节点下的 URL 地址
```

`visit` 方法通过从网页根节点递归遍历返回符合条件的 DIV 节点。

```
public static Node visit(Node node) {
    int type = node.getNodeType();

    if (type == Node.ELEMENT_NODE && "DIV".equals(node.getNodeName())) {
        Node att = node.getAttributes().getNamedItem("style");
        if (att != null
            && "float:left;width:464px;margin:1px;border: 1px solid #ccc;"
                .equals(att.getNodeValue())) {
            return node;
        }
    }

    Node child = node.getFirstChild();
    while (child != null) {
        //递归调用
        Node t = visit(child);
    }
}
```

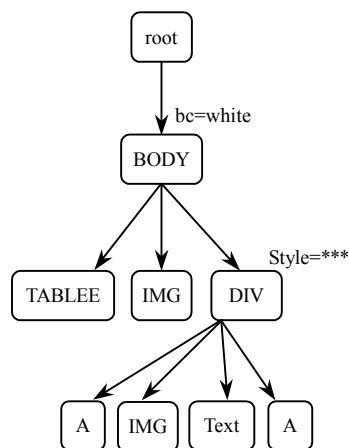


图 3-8 选取 DIV 下的节点



```
    if (t != null) {
        return t;
    }
    child = child.getNextSibling();
}
return null;
}
```

visitDiv 方法通过从 DIV 节点递归遍历，打印出 DIV 节点下的 URL 地址。

```
public static void visitDiv(Node node){
    int type = node.getNodeType();

    if (type == Node.ELEMENT_NODE && "A".equals(node.getNodeName())) {
        Node t = node.getAttributes().getNamedItem("title");
        if(t!=null) {
            String title = t.getNodeValue();
            Node h = node.getAttributes().getNamedItem("href");
            String url = h.getNodeValue();
            System.out.println(title+"\t"+url);
        }
    }

    Node child = node.getFirstChild();
    while (child != null) {
        //递归调用
        Node t = visitDiv(child);
        child = child.getNextSibling();
    }
}
```

提取“更多”文本对应的链接：

```
public static String getMoreUrl(Node node) throws Exception {
    int type = node.getNodeType();
    if(type == Node.TEXT_NODE && "更多".equals(node.getNodeValue())) {
        return node.getParentNode().getAttributes().getNamedItem(
            "href").getNodeValue();
    }

    Node child = node.getFirstChild();
    while (child != null) {
        //递归调用
        String t = getMoreUrl(child);
        if (t != null) {

```

```

        return t;
    }
    child = child.getNextSibling();
}
return null;
}

```

下面的代码移除节点 iNode:

```
iNode.getParentNode().removeChild(iNode);
```

为了避免乱码问题,可以先下载整个网页缓存到字符串,然后再解析,具体代码如下所示:

```

//content 是包含网页内容的字符串
InputSource inputsource=new InputSource(new StringReader(content));
parser.parse(inputsource);

```

### 3.1.7 使用 Jsoup 提取信息

Jsoup (<http://jsoup.org/>) 和 NekoHTML 一样,能够根据不规范的 HTML 格式生成 DOM 树。补全有开始没结束的标签。例如,把 `<p>Lorem <p>Ipsum` 解析成 `<p>Lorem</p><p>Ipsum</p>`。这里的两个 `</p>` 是等价标签。

NekoHTML 需要 Xerces,而 Jsoup 没有依赖其他的 jar 包。例如只需要 jsoup-1.6.3.jar 这一个 jar 包。在爬虫项目中增加对 jsoup-1.6.3.jar 的引用。

Jsoup.connect(String url)方法创建一个新的连接。get()方法取得并解析一个 HTML 文件。如果发生错误,就抛出一个 IOException。从一个 URL 解析 HTML 的代码:

```

String url = "http://www.lietu.com";
Document doc = Jsoup.connect(url).get();

```

设置连接参数:

```

Document doc = Jsoup.connect("http://www.lietu.com/")
    .data("query", "Java") //请求参数
    .userAgent("jsoup") //设置 User-Agent
    .cookie("auth", "token") //设置 cookie
    .timeout(3000) //设置连接超时时间
    .post(); //使用 POST 方法访问 URL

```



也可以从字符串中提取：

```
String html = "<html><head><title>First parse</title></head>"
    + "<body><p>Parsed HTML into a doc.</p></body></html>";
Document doc = Jsoup.parse(html);
```

如果本地硬盘中的一个文件缓存了 HTML 页面，可以加载这个文件并解析内容：

```
File input = new File("d:/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://www.lietu.com");
```

得到标题：

```
Document doc = Jsoup.connect("http://www.lietu.com/").get();
String title = doc.title(); //取出网页的标题
```

Element.text()方法输出节点对应的文本。例如，对于 HTML 文本 `<p>Hello <b>there</b>now!</p>`，调用 p.text()方法，返回"Hello there now!"。

Element.html()返回这个节点代表的整个 HTML，会保留换行符。Element.outerHtml()则会返回包含描述 Element 标签在内的所有 HTML 内容。

Jsoup 支持使用 DOM 来查找、取出数据。可以使用 CSS 选择器来查找、取出数据。例如，选出带有 href 属性的 a 类型的标签：

```
Elements links = doc.select("a[href]"); //带有 href 属性的 a 标签
```

可以提取下面这个 a 标签中的网址：

```
<a href="/www/detail/detail.jsp?DOCID=16951" target="_blank" title="
企业资产损失所得税税前扣除相关表单">
```

提取 a 标签中的 title 属性包括了“讲话”字样：

```
Elements links = doc.select("a[title~=讲话]");
```

提取网页中链接的完整例子如下：

```
String url = "http://www.lietu.com";
Document doc = Jsoup.connect(url).get();
Elements links = doc.select("a[href]"); //带有 href 属性的 a 标签
for (Element link : links) { //遍历每个链接
```

```
String linkHref = link.attr("href"); //得到 href 属性中的值, 也就是 url 地址
String linkText = link.text(); //得到锚点上的文字说明
System.out.println(linkHref+" "+linkText); //输出 url 地址和锚点上的文字说明
}
```

除了 CSS 选择器, Jsoup 还提供了类似于 jQuery 的操作方法来取出和操作数据。getElementById 和 getElementsByTag 方法跟 JavaScript 中的方法名称是一样的, 功能也完全一致。可以根据节点名称或者 HTML 元素的 id 来获取对应的元素或者元素列表:

```
File input = new File("D:/test.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://www.lietu.com/");

Element content = doc.getElementById("content"); //通过 id 名称获取对应元素
Elements links = content.getElementsByTag("a"); //通过类型获取元素列表
for (Element link : links) { //遍历每个链接
    String linkHref = link.attr("href"); //得到 href 属性中的值, 也就是 url 地址
    String linkText = link.text(); //得到锚点上的文字说明
}
```

可操作 HTML 元素、属性、文本。

```
for (Element link : links) {
    System.out.println(" * a: <%s> (%s)", link.attr("abs:href"),
link.text());
}
```

可以通过 class 或者 id 信息选取元素。例如, 选择目录链接所在的 Div 区域<div class='div.masthead'>:

```
//通过 class 选取元素
Element mainDiv = doc.select("div.masthead").first();
```

想要选择<div id="content">, 可以使用 CSS ID 选择器采用格式"#id":

```
Document document = Jsoup.connect("http://www.qualcomm.com/innovation").get();
Element content = document.select("#content").first();
System.out.println(content.html());
```

"<!-- -->"是网页中的注释标签。注释是节点, 用节点名#comment 标识。去掉注释节点的代码如下:

```
public class RemoveComments {
```





```
public static void main(String... args) {
    String h = "<html><head></head><body>" +
        "<div><!-- foo --><p>bar<!-- baz --></div><!-- qux--></body>"
    </html>";

    Document doc = Jsoup.parse(h);
    removeComments(doc);
    System.out.println(doc.html());
}

private static void removeComments(Node node) {
    for (int i = 0; i < node.childNodes().size()) {
        Node child = node.childNodes().get(i);
        if (child.nodeName().equals("#comment"))
            child.remove();
        else {
            removeComments(child);
            i++;
        }
    }
}
```

每一个节点在 DOM 树中都有一个特定的位置。Node 接口中有一些发现一个指定节点的周边节点的方法。

Jsoup 提供了图形化方式导航：**parent()**得到父亲节点、**children()**得到所有的孩子节点、**child(int index)**得到指定的孩子节点。

如下面代码：

```
string html = "<div id='demo'><span style='color:red;'><h1>Hello World!</h1></span><div id='innerDiv'>inner div</div></div>";
```

**FirstChild** 属性返回所有子节点的第一个节点。这里 **FirstChild** 返回的是 “<span style='color:red;'><h1>Hello World!</h1></span>” 的节点。

**LastChild** 属性返回所有子节点的最后一个节点。这里 **LastChild** 返回 “<div id='innerDiv'>inner div</div>” 节点。

**children()**返回当前节点所有直接一代的子节点的集合，不包括跨代子节点，这里返回 “<span style='color:red;'><h1>Hello World!</h1></span>” 和 “<div id='innerDiv'>inner

div</div>”两个节点。

NodeVisitor接口用来遍历DOM树中的节点。这个接口提供了两个方法,一个叫做head,另外一个叫做tail。当第一次看到节点时,调用head方法。当这个节点所有的孩子节点都已经访问过以后,调用tail方法。例如,可以使用head创建一个节点的开始标签,用tail创建结束标签。

```
Document doc = Jsoup.parse(html);

//用一个 NodeVisitor 对象构造一个 NodeTraversor
NodeTraversor nd = new NodeTraversor(new NodeVisitor() {

    @Override
    public void tail(Node node, int depth) {
        //处理代码...
    }

    @Override
    public void head(Node node, int depth) {
    }

});

nd.traverse(doc.body());
```

### 3.1.8 网页去噪

一个页面中经常包括导航栏或底部的公司介绍等信息,这样的信息在很多页面都会出现,可以看作噪音信息。一般情况下可以去掉网页DOM树中FORM、Select、IFRAME、INPUT、STYLE的节点。

不同的页面类型可以使用不同的去噪方法。常见的两种网页类型是目录导航式页面(List Page)和详细页面(Detail Page)。详细页面需要抽取的正文信息包括标题和内容等。

详细页面的特征有以下几点。

- 非锚点文本较多。
- 一般都有明显的文本段落,文字较多,相应的标点符号也较多。
- URL较长。在一般的Web网站链接导航树上,主题型网页主要分布于底层,多为叶节点。对于同一网站而言,主题型网页的URL相对较长。URL体现了网站内容管理的层次,对于大型网站而言,URL往往非常有规律。



- 链接较少。主题型网页的主体在于“文字”，相对于导航型网页，其链接数较少。
- 主体文字在 DOM 树中的层次较深。
- 网页标题可能出现在 Title 标签或 H1 标签中，H2 标签可能表示文章的段落标题。

详细页面中网页噪音特征有以下几点。

- 多以链接的形式出现，链接到别的相关页面。
- 有很多锚文本，但标点符号较少，锚文本往往是对其他链接页面的说明。
- 有许多常见的噪音文本，如版权声明等。在视觉上，多出现于网页的边缘。

网页去噪的方法基本是利用各种通用的特征来区分有效的正文和页眉、页脚、广告等其他信息。其中一个常用的特征是链接文字比率。可以把 HTML 转换成 DOM 树，对每个节点计算链接文字比率。如果该节点的链接文字比率高于 1/4，就把这个节点去掉。

下面使用递归调用的方法计算一个节点下的链接数。

```
/**
 * 计算一个节点下的链接数
 *
 * @param iNode 开始计算的节点
 * @return 链接数
 */
public static int getNumLinks(final Node iNode) {
    int links = 0;
    if (iNode.hasChildNodes()) {
        Node next = iNode.getFirstChild();
        while (next != null) {
            Node current = next;
            next = current.getNextSibling();
            //递归调用计算链接数的方法
            links += getNumLinks(current);
        }
    }

    if (isLink(iNode))
        links++;

    return links;
}
```

计算一个节点下的有效正文长度，忽略锚点上的字。

```

/**
 * 计算一个节点下的单词数
 *
 * @param iNode 开始计算的节点
 * @return 单词数
 */
private int getNumWords(final Node iNode) {
    int words = 0;

    if (iNode.hasChildNodes()) {
        Node next = iNode.getFirstChild();

        while (next != null) {
            Node current = next;
            next = current.getNextSibling();

            //如果当前节点是一个链接，则不往下深入
            if (!isLink(current))
                words += getNumWords(current);
        }
    }

    //检查节点是文本节点还是元素节点
    int type = iNode.getNodeType();

    //文本节点
    if (type == Node.TEXT_NODE) {
        String content = iNode.getNodeValue();
        words += getHTMLLen(content);
    }

    return words;
}

```

计算一个文本中大概包括的正文实际长度的方法如下所示：

```

private int getHTMLLen(String text) {
    int len = 0;
    for(int i=0;i<text.length();++i) {
        if(text.charAt(i)==' ') {

        }
        else if(text.charAt(i)==' ') {

        }
    }
}

```



```
        else if(text.charAt(i)==' ')    {

        }
        else if(text.charAt(i)=='&')    {
            i+=5;
        } else {
            ++len;
        }
    }
    if( len<10)
        len = 0;
    return len;
}
```

根据链接文字比删除无效节点的过程说明如下。

- ① 计算节点下的链接数。
- ② 计算节点下的文字数。
- ③ 计算节点的“链接文字比 = 节点下的链接数 / 节点下的文字数”。
- ④ 如果节点的链接文字比大于某一个阈值则删除这个节点。

删除无效节点的实现代码如下所示：

```
/**
 * 如果链接比率合适则删除这个表单元
 * @param iNode 表单元节点
 */
public void testRemoveCell(final Node iNode) {
    //如果这个表单元没有孩子节点则不处理这个表单元
    if (!iNode.hasChildNodes())
        return;

    double links;//iNode 节点下的链接数
    double words;//iNode 节点下的单词数

    //计算链接和单词数
    links = getNumLinks(iNode);
    words = getNumWords(iNode);

    //计算链接文字比并检查是否被 0 除
    double ratio = 0;
```

```

if (words == 0)
    ratio = settings.linkTextRatio + 1;
else
    ratio = links / words;

if (ratio > settings.linkTextRatio) { //如果链接文字比大于指定值则删除该节点
    iNode.getParentNode().removeChild(iNode);
}
}

```

NekoHTML 可以把 HTML 文档转换成 XML 文档。XML 文档的某一部分可以用 XPath 来描述。可以用计算节点特征的方法找到覆盖主要正文的内容节点。找到这样一个节点后，可以把这个节点用 XPath 表示出来，例如：

```

/HTML[1]/BODY[1]/DIV[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY
[1]/TR[1]/TD[1]

```

但是这种 XPath 的绝对路径表示方式当 DOM 树的节点有删除或修改后就失效了。

为了支持通过 XPath 取得节点，Java 项目必须导入 xalan.jar。下面的例子提取当当网图书的 ISBN 信息：

```

DOMParser parser = new DOMParser();

parser.setProperty("http://cyberneko.org/html/properties/default-enc
oding", "gb2312");
parser.setFeature("http://xml.org/sax/features/namespace", false);
String bookURL =
    "http://product.dangdang.com/product.aspx?product_id=20733895";
parser.parse(bookURL);

Document doc = parser.getDocument();
String isbnXPath = "/HTML/BODY/DIV[3]/DIV[6]/DIV[2]/DIV/DIV[3]/UL/LI[9]";
NodeList products;
products = XPathAPI.selectNodeList(doc, isbnXPath);
System.out.println("found: " + products.getLength());
Node node = null;
for (int i = 0; i < products.getLength(); i++) {
    node = products.item(i);
    System.out.println(i + ":\n" + node.getTextContent());
}

```

### 3.1.9 网页结构相似度计算

自动提取结构化信息的关键是从同样类型的实例中发现编码模版。为了确定两个网页是否由同一个网页模板生成,需要计算两个网页的结构相似度。一个自然的方法是从 HTML 编码字符串检测重复的模式。检测的方法有最长公共子序列和树编辑距离。

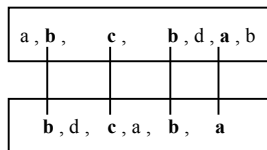


图 3-9 最长公共子序列

举例说明两个序列  $s_1$  和  $s_2$  的最长公共子序列 (Longest Common Subsequence, LCS)。  $s_1 = \{a, b, c, b, d, a, b\}$ ,  $s_2 = \{b, d, c, a, b, a\}$ , 则从前往后找,  $s_1$  和  $s_2$  的最长公共子序列为  $LCS(s_1, s_2) = \{b, c, b, a\}$ , 如图 3-9 所示。

LCS 问题的最优解只取决于其子序列 LCS 问题的最优解, 而非最优解对问题的求解没有影响。同时子序列 LCS 的值和当前对比字符的值一旦确定, 则此后过程的 LCS 计算不再受此前各状态及决策的影响。因为满足动态规划的最优化原理和无后效性原则, 因此使用动态规划的思想计算 LCS 的方法是: 引进一个二维数组  $num[i][j]$ , 用  $num[i][j]$  记录  $s_1$  的前  $i$  个长度的子串与  $s_2$  的前  $j$  个长度的子串的 LCS 长度。

自底向上进行递推计算, 那么在计算  $num[i][j]$  之前,  $num[i-1][j-1]$ 、 $num[i-1][j]$  与  $num[i][j-1]$  均已计算出来。此时再根据  $s_1[i-1]$  和  $s_2[j-1]$  是否相等, 就可以计算出  $num[i][j]$ 。

采用动态规划的方法计算两个序列的最长公共子序列的实现代码如下所示:

```
public static <E> List<E> longestCommonSubsequence(E[] s1, E[] s2){
    int[][] num = new int[s1.length+1][s2.length+1]; //二维数组

    //实际算法
    for (int i = 1; i <= s1.length; i++)
        for (int j = 1; j <= s2.length; j++)
            if (s1[i-1].equals(s2[j-1]))
                num[i][j] = 1 + num[i-1][j-1];
            else
                num[i][j] = Math.max(num[i-1][j], num[i][j-1]);

    System.out.println("length of LCS = " + num[s1.length][s2.length]);

    int s1position = s1.length, s2position = s2.length;
    List<E> result = new LinkedList<E>();

    while (s1position != 0 && s2position != 0) {
```

```

        if (s1[s1position - 1].equals(s2[s2position - 1])) {
            result.add(s1[s1position - 1]);
            s1position--;
            s2position--;
        }
        else if
            (num[s1position][s2position - 1] >= num[s1position - 1]
            [s2position]) {
            s2position--;
        }
        else {
            s1position--;
        }
    }
    Collections.reverse(result);
    return result;
}

```

比较网页结构相似度的基本流程是：首先把网页抽象成一个 Node 数组，然后比较两个 Node 数组的最长公共子序列。输入两个 URL 地址，返回网页相似度的实现代码如下所示：

```

public static double getPageDistance(String urlStr1,String urlStr2) {
    ArrayList<Node> pageNodes1 = new ArrayList<Node>();//网页转换成 Node 数组

    URL url = new URL(urlStr1);
    Node node;
    Lexer lexer = new Lexer (url.openConnection ());
    lexer.setNodeFactory(new PrototypicalNodeFactory ());
    while (null != (node = lexer.nextNode ())) {
        pageNodes1.add(node);
    }

    ArrayList<Node> pageNodes2 = new ArrayList<Node>();
    URL url2 = new URL(urlStr2);

    lexer = new Lexer (url2.openConnection ());
    lexer.setNodeFactory(new PrototypicalNodeFactory ());
    while (null != (node = lexer.nextNode ())) {
        pageNodes2.add(node);
    }

    int lcs = longestCommonSubsequence (pageNodes1, pageNodes2);
    return lcs / (double)Math.min(pageNodes1.size(), pageNodes2.size());
}

```



### 3.1.10 提取标题

有两种提取标题的方式：一种是利用锚点描述文字；还有一种是利用标题中的描述信息。例如：

<title>中国燃气总经理和执行总裁疑被警方带走\_网易新闻中心</title>

<title>干部考核不宜唯“德孝”是举 - 第一视点 - 华声评论 - 华声在线</title>

在“-”或“\_”等分隔符前面的文字是真正的标题。

网页链接中的锚点文本是对目标网页主题内容的概括，所以往往用它作为一个网页的标题描述。图 3-10 显示了来源于新华网的两个页面之间的对应关系：

先把列表页上的链接 URL 对应的 <a> 标签的“title”属性获取下来，称为列表页描述标题。详细页面的标题属性称为内部标题。比较列表页描述标题和内部标题，估计出详细页的标题。由于这两个标题的标题字符串不是全部相同，例如列表页描述标题如果较长就可能以省略号结尾或者末尾截短字，而内部标题可能包含“\_网站名”等冗余信息。所以可以考虑通过最长公共子串的方式比较两个标题进行相似度判断。

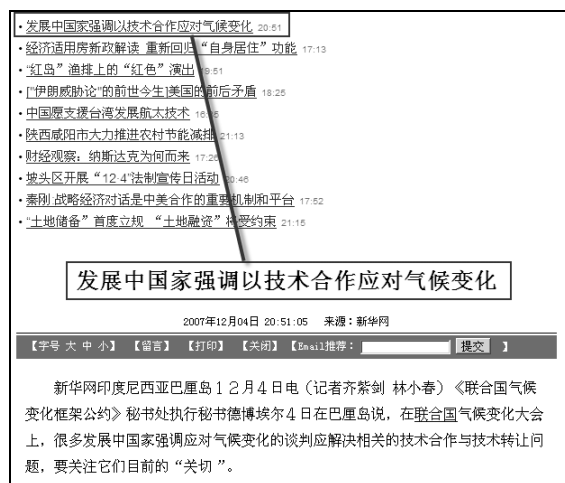


图 3-10 网页标题和锚点文本的对应关系

取得了网页标题以后，还可以利用标题信息计算网页的内容和标题之间的距离。

```
public static double getSimiarity(String title,String body){
    int matchNum = 0;
    for(int i=0;i<title.length();++i) {
        if(body.indexOf(title.charAt(i))>=0) {
            ++matchNum;
        }
    }
    double score = (double)matchNum/( (double)title.length() );
    return score;
}
```

可以对每个文本类型的节点根据内容分类，例如标题、正文、来源、控制页面的文本、广告文本、版权信息文本等。

把所有文本类型的节点收集到一个数组中，然后对每一个节点应用规则提取内容，并且组织成树形结构，最后再按节点解析成一个大的树形结构。

### 3.1.11 提取日期

经常需要从网页提取日期来判断网页的新旧程度，例如“2009-12-6”这样的格式可以用 `java.util.regex` 包中的正则表达式匹配如下：

```
Pattern p = Pattern.compile("\\d{2,4}-\\d{1,2}-\\d{1,2}");
Matcher m = p.matcher(inputStr);
if(m.find()){
    String strDate = m.group();
}
```

其他的一些正则表达式模式有：“`\\d{2,4}/\\d{1,2}/\\d{1,2}`”和“`\\d{2,4}年\\d{1,2}月\\d{1,2}日`”以及“`\\d{2,4}\\.\\d{1,2}\\.\\d{2,4}`”。

也可以用有限状态机提取日期。

很多新闻网站的新闻都是按发布日期分目录存放的，例如 `http://finance.sina.com.cn/g/20101226/09339163619.shtml` 是 2010 年 12 月 26 日的新闻。所以还可以从 URL 地址提取日期。



## 3.2 从非 HTML 文件中提取文本

在很多知识库系统中，为了查询大量积累下来的文档，需要从 PDF、Word、Rtf、Excel 和 PowerPoint 等格式的文档中提取可以描述文档的文字。其中 Word、Rtf、Excel 和 PowerPoint 格式都来自微软。POI 项目 (`http://poi.apache.org/`) 是一个专门处理微软文档格式的开源项目，是一个纯 Java 的实现，可以在 Linux 平台运行。S1000D 是一种 XML 文档格式，主要用于航空航天和国防业（军品和民品）中的技术出版物。

有很多文件名的命名没有意义，比如说类似“20090224153208138.pdf”由数字组成的文件名。在搜索结果中显示一个有意义的文件标题而不是文件名能够改进用户的搜索体验。

这里首先从一般意义上来讨论从文件内容提取标题的方法，然后按文件类型分别具体实现提取标题。

设计一个通用的接口来处理待索引的文档。

```
//处理文档
public interface IFilter {
    String getTitle(File file); //返回标题
    String getBody(File file); //返回内容
    IDocument getDocument(File file); //返回全部索引信息
}
```

### 3.2.1 提取标题的一般方法

为了从正文中比较合理地提取标题，设计提取标题流程如图 3-11 所示。

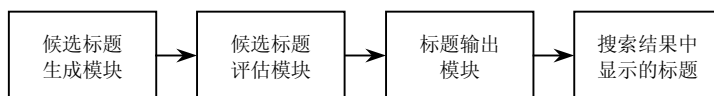


图 3-11 提取标题流程

- 候选标题生成模块：首先将提取出来的原子文本组织成文档结构树。构建文档结构树既可以用自底向上的方法，从原子节点构造起，也可以用自顶向下的方法从根节点首先将文字划分成大的单元，然后逐步从大块文字细分。如果采用自底向上的方法，文字在同一行，并且字体、字号、颜色都一致，则视为不可拆分。然后根据字体、字号、颜色、位置等信息再次合并文字，从文档结构树给出几个可能的候选标题。
- 候选标题评估模块：对每个候选标题按照特征打分。可以把要用到的特征都作为候选标题类的属性。其中，最主要的特征是标题字符串，此外还有字符串所在位置、字号、字体颜色等信息。可以根据标题字符串对整个文章的概括程度和通顺性与意义完整性打分。从对文章的概括程度考虑，可以按照  $TF*IDF$  等方法选取重要性较高的词作为关键词，然后根据关键词对每个候选标题字符串给出可能性权重。还可以比较候选标题字符串和首页中的其他文字，看候选标题相对其他文字的代表度或者说是相对其他文字的可替代性，也就是说候选标题对其他文字的覆盖度。从通顺性与意义完整性考虑，可以考虑准备一个标题语料库，提取出语法规则和作为标题常用的搭配规则，也可以对大量标题训练一个 HMM 模型，或者用信息提取的方法来评估候选标题字符串。

- 标题输出模块：把权重最大的候选标题挑出来，按照可读的方式输出。

作为标题的文字长度往往既不是太长，也不会太短，大部分标题的长度在 10 到 30 个字之间。超长的标题往往会被搜索引擎截短，网站制作者为了优化搜索排名，也倾向于采用这样的标题长度。可以用 `sweetSpot` 来对候选标题的长度打分。

```
private static int ln_min = 10; //标题最短长度
private static int ln_max = 30; //标题最长长度
private static float ln_steep = 0.5f;

//如果长度在 10 到 30 之间，则 sweetSpotScore 返回值是 1，
//否则返回值会低于 1，返回值随长度降低的程度由 ln_steep 决定
public static double sweetSpotScore(int length) {
    return (float) (1.0f /
        Math.sqrt((ln_steep * (float) ( Math.abs(length - ln_min)
            + Math.abs(length - ln_max) - (ln_max - ln_min)) ) + 1.0f) );
}
```

`ln_steep` 一般取值在 0 到 1 之间，`ln_steep` 越小，则返回值下降幅度越小。

政府公文有比较固定的格式，有“发文单位”、“文号”、“接收单位”等。如图 3-12 所示的文件“发文单位”是“合肥市物价局文件”；“文号”是“合价服〔2009〕219 号”；“接收单位”是“三县四区物价局”。



图 3-12 政府公文

判断是否为发文单位的代码如下所示：

```
public static boolean isProSendGovUnit(String s, Color tcolor){
    double pro = 0;

    if(tcolor.equals(Color.RED))    {
        pro+=0.2;
    }

    if(s.endsWith("文件")){
        pro+=0.4;
    }

    int numBlank = 0;
    int numGovUnit = 0;
    for(int i=0; i<s.length(); i++){
```



```
        char c = s.charAt(i);
        if(c == ' '){
            numBlank ++;
        }
        else if(c == '厅' ||
                c == '所' ||
                c == '局' ||
                c == '会' ||
                c == '委' ||
                c == '府')
            numGovUnit++;
    }
    if(numBlank > 0)    {
        pro += (0.6*((double)numGovUnit+(double)numBlank)/((double)
            numBlank+s.length()));
    }
    if(pro > 0.6)
        return true;
    return false;
}
```

判断是否为接收单位的代码如下所示：

```
public static boolean isProReceiGovUnit(String s,Color tcolor){
    if(!tcolor.equals(Color.BLACK)) {
        return false;
    }

    float pro =0 ;
    if(s.endsWith(": ") || s.endsWith(":")) {
        pro += 0.5;
    }

    for(int i=0; i<s.length(); i++) {
        char c = s.charAt(i);
        if(c == ',' || c == '(' || c == ')' || c == ',')    {
            pro +=0.2;
        }
        else if(c == '厅' ||
                c == '部' ||
                c == '所' ||
                c == '局' ||
                c == '会' ||
                c == '委' ||
                c == '县' ||
```

```

        c == '区' ||
        c == '市' ||
        c == '省')
        pro +=0.11;
    }
    pro = pro/s.length();
    if(pro >= 0.07)
        return true;
    return false;
}

```

判断文号的代码如下所示:

```

public static boolean isFileNum(String s,Color tcolor){
    if(!tcolor.equals(Color.BLACK)){
        return false;
    }
    boolean isSymbol = false;

    //如果碰到开始符号,则匹配对应的结束符号
    for(int index=0; index<s.length(); index++){
        if('[' == s.charAt(index)){
            isSymbol = matchSym(s,index,']');
        }
        else if('(' == s.charAt(index)) {
            isSymbol = matchSym(s,index,')');
        }
        else if ('{' == s.charAt(index)) {
            isSymbol = matchSym(s,index,'}');
        }
    }

    return isSymbol;
}

//如果匹配上指定字符则成功退出
public static boolean matchSym(String s,int index,char endMark){
    boolean isSymbol = false;

    boolean markBegSym = true;
    boolean markEndSym = false;
    for(int i =index+1; i<s.length(); i++) {
        if(!markEndSym) {
            if((s.charAt(i)>='0'&& s.charAt(i)<='9')||s.charAt(i) == ' '
                ||(s.charAt(i)>='0'&& s.charAt(i)<='9'))

```



```
    {}
    else if(s.charAt(i) == endMark)    {
        markEndSym = true;
        i++;
    }
    else {
        markBegSym = false;
        markEndSym = false;
    }
}
if(markBegSym && markEndSym)    {
    if((s.charAt(i)>='0'&& s.charAt(i)<='9')||s.charAt(i) == ' '
        ||(s.charAt(i)>='0'&& s.charAt(i)<='9'))
    {}
    else if(s.charAt(i) == '号')    {
        isSymbol = true;
        index = s.length();
    }
}

return isSymbol;
}
```

### 3.2.2 PDF 文件

PDF 是 Adobe 公司开发的电子文件格式。这种文件格式与操作系统的平台无关，可以在多数操作系统上通用。我们经常会遇到这种情况，就是想把 PDF 文件中的文字复制下来，但是发现经常不能复制，因为这个 PDF 文件的内容可能加密了。那么怎么把 PDF 文件中的内容提取出来？这就是我们这一节要解决的问题。现在已经有好多工具能够完成这项任务了，PDFBox（<http://pdfbox.apache.org/>）就是专门用来解析 PDF 文件的 Java 项目。

下载文件 pdfbox-app-1.4.0.jar，然后在 Eclipse 项目中引入这个文件。提取文本只需要以下三行代码：

```
//加载 fileName 代表的 PDF 文件
PDDocument doc = PDDocument.load(fileName);
//用 PDFTextStripper 提取文本
PDFTextStripper stripper = new PDFTextStripper();
//返回提取的文本字符串
String content = stripper.getText(doc);
```

这样可以抽取 PDF 文件中的文本，但是对于 PDF 文件中的图片 PDFBox 是无能为力

的。对于包含文字的图片，需要借助 OCR 软件从图片中识别出文字。

需要说明的是这个版本部分支持中文。使用 PDFBox 抽取中文 PDF 文本时，可能会遇到 Unknown encoding for 'GBK-EUC-H'错误。因为 PdfReader 内部默认支持一些中文字体，但 PDFBox 内部却没有对'GBK-EUC-H'字体的支持。

在 PDFBox 的 0.8 版本中，会查找操作系统本身的字体。

```
java.awt.Font[] allFonts =
    java.awt.GraphicsEnvironment.getLocalGraphicsEnvironment().
        getAllFonts();
int numberOfFonts = allFonts.length;
for (int i=0;i<numberOfFonts;i++) {
    java.awt.Font font = allFonts[i];
    System.out.println(font);
}
```

但是'GBK-EUC-H'是PdfReader内部自带的字体，而不是操作系统支持的true type字体，所以这种方法仍然找不到'GBK-EUC-H'字体。PDFBox的751690版本支持类似'GBK-EUC-H'的字体，可以用SVN客户端把751690版本从<http://svn.apache.org/repos/asf/incubator/pdfbox/trunk/>下载下来，然后用Ant编译一下即可。

因为PDF文件中有可能存在重叠位置的文字，PDFTextStripper有对于重叠位置文字的判断。如果两个TextPosition位于差不多同样的位置，则不重复抽取重叠TextPosition代表的文字。

```
private boolean overlap( TextPosition tp1, TextPosition tp2 ){
    if(!within(tp1.getHeight(),tp2.getHeight(), 1.1f))
        return false;

    float diff = (tp1.getHeight() + tp2.getHeight())*0.02f;
    return within( tp1.getX(), tp2.getX(), diff) && within( tp1.
        getY(), tp2.getY(), diff) ;
}
```

在提取了PDF文件中可搜索的文字后，下一个问题是提取标题。例如，从Google搜索“ filetype:pdf”可以看到Google提取的PDF文件标题。

如果PDF元数据中已经存储了文档标题，可以通过元数据取得文档标题：





```
//取得文档元数据
PDDocumentInformation info = document.getDocumentInformation();
this.title = info.getTitle();
```

但是很多 PDF 文件中元数据并没有存储任何内容。所以在大多数情况下，仍然需要从内容中分析出标题。

可以利用文本的位置或字体等信息帮助选取标题。例如在首页中字体最大的文字有可能是 PDF 文件的标题，可以通过 `TextPosition` 对象的 `getFontSizeInPt` 方法返回字体的大小。

为了取得文本的位置信息和颜色等，需要更加深入地了解 `PDFBox` 的运行原理。PDF 规范可以从 [http://www.adobe.com/devnet/pdf/pdf\\_reference.html](http://www.adobe.com/devnet/pdf/pdf_reference.html) 下载。PDF 内容流中包含许多操作符（Operator）。在 `PDFBox` 中，每种操作符都有专门对应的处理类，并记录在配置文件 `Resources/PageDrawer.properties` 中。例如其中的一行：

```
BT=org.apache.pdfbox.util.operator.BeginText
```

表示 BT 操作的处理类是 `org.apache.pdfbox.util.operator.BeginText`。

`PDFBox` 中的应用程序 `PageDrawer` 实现输出 PDF 文件到可视化窗口。`PageDrawer` 有完整的对于文字显示方面的处理。为了提取标题，可以通过在 `PageDrawer` 的实现基础上，简化一些与 `Path` 和 `Line` 相关的操作符的处理，只把位置信息和颜色等信息提取出来。

文字的颜色信息并不能直接得到，而是依赖于上下文。`PageDrawer` 类包含了对图像的处理。

```
if( this.getGraphicsState().getTextState().getRenderingMode() ==
    PDTextState.RENDERING_MODE_FILL_TEXT ){
    graphics.setColor( this.getGraphicsState().getNonStroking
        ColorSpace().createColor() );
}
else if( this.getGraphicsState().getTextState().getRenderingMode() ==
    PDTextState.RENDERING_MODE_STROKE_TEXT ) {
    graphics.setColor( this.getGraphicsState().getStroking
        ColorSpace().createColor() );
}
```

用 `PdfTitle` 表示候选标题。

```
public class PdfTitle {
```

```

public String title = null; //标题文字
public Color textColor = null; //文字颜色
public float fontSize; //字体大小
public float y; //高度
ArrayList<TextPosition> texts; //包含的文本

public PdfTitle(String t,float h) {
    title = t;
    y=h;
}

public PdfTitle(Color c,float f,float h,ArrayList<TextPosition> text) {
    title = delOverlap(text);
    textColor = c;
    fontSize = f;
    y = h;
    texts = text;
}
}

```

候选标题单独组成一个段落，或者位于一个段落内部。通过 PDFDegger 可以看到 PDF 文件的树形结构组织。因为通过 PDFStreamEngine 类导出的文字块没有明显的段落信息，所以需要编写程序寻找段落边界。大的字体差别和颜色差别以及文本的垂直位置都可以用来确定是一个独立段落的开始，但有些相对模糊的边界需要更细致的判断。因此把 PdfTitle 设计成可以再次拆分的单元。

- ① 根据文本的垂直位置寻找该段落的最大垂直区隔。
- ② 根据最大垂直区隔拆分 PdfTitle。
- ③ 如果没有找到合适的标题，对于拆分出来的新的 PdfTitle 重新应用①，②直到不可再拆分或找到合适的标题为止。

可以通过扩展 org.pdfbox.util.PDFTextStripper 类和覆盖 processTextPosition 方法来遍历 PDF 文件中的 TextPosition 对象。下面的代码简单地提取最大字号的文字作为标题。

```

float currentFontSize = text.getFontSizeInPt();
if(currentFontSize < biggestFontSize){ //字号不一致，则不是标题文字
    consistent = false;
} else if (currentFontSize > biggestFontSize){ //字号最大，是标题文字
    titleGuess = text.getCharacter();
    biggestFontSize = currentFontSize;
}

```



```

        consistent = true;
    } else if(currentFontSize == biggestFontSize
        && consistent
        && !"".equals(text.getCharacter().trim())){
        //字号和以前文字的最大字号一样大，是标题文字
        titleGuess += text.getCharacter();
    }
}

```

PDF 文件分成两类：一类首页没有正文，文字比较少，文字比较多的部分可能是标题；还有一类，首页有正文，挨着正文往上的可能是标题，也可能是段落标题。首页没有正文的情况，不计算标题对于首页正文的概括性。

政府公文中往往包含一些主题词，可以利用主题词或者文章内容来判断标题。或者建立一个标题的语料库，例如很多标题都是采用“关于\*\*的通知”这样的形式，可以利用模版匹配的方式来保证标题的完整性。

### 3.2.3 Word 文件

Word 是微软公司开发的字处理文件格式，以“doc”或者“docx”作为文件后缀名。Apache 的 POI(<http://poi.apache.org/>)可以用来在 Windows 或 Linux 平台下提取 Word 文档。用 POI 提取文本的基本方法如下：

```

public static String readDoc(InputStream is) throws IOException{
    //创建 WordExtractor
    WordExtractor extractor=new WordExtractor(is);
    //对 DOC 文件进行提取
    return extractor.getText();
}

```

为了提取 Word 文档的标题，需要深入了解 POI 接口。一个 Word 文档包含一个或者多个 Section，每个 Section 下面包含一个或者多个 Paragraph，每个 Paragraph 下面包含一个或者多个 CharacterRun，如图 3-13 所示。

遍历 Word 文档结构的代码如下所示：

```

Range r = doc.getRange();

for (int x = 0; x < r.numSections(); x++) {

```

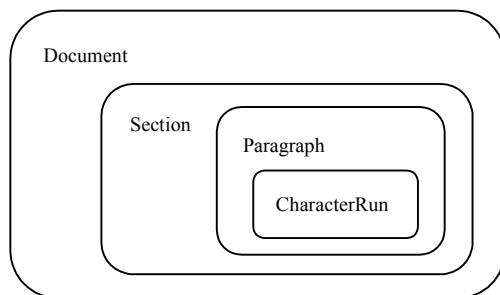


图 3-13 Word 文档结构图

```

Section s = r.getSection(x);
for (int y = 0; y < s.numParagraphs(); y++) {
    Paragraph p = s.getParagraph(y);
    for (int z = 0; z < p.numCharacterRuns(); z++) {
        CharacterRun run = p.getCharacterRun(z);
        //字符串文本
        String text = run.text();
        System.out.println(text);
    }
}
}

```

标题往往是居中对齐的。可以通过 Paragraph 的 getJustification 方法得到段落的对齐方式。getJustification 的返回值为 0 表示左对齐；1 表示居中对齐；2 表示右对齐；3 表示两端对齐。

可以通过 CharacterRun 对象取得文字内容、字号、文字颜色等信息。

```

run.text();    //文字内容
run.getFontSize(); //字号
run.getColor(); //文字颜色

```

### 3.2.4 Rtf 文件

1992 年,微软公司为了定义简单的格式化文本和嵌入的图片引入了富文本格式(RTF)。最开始是为了在不同的操作系统(例如 MS-DOS、Windows 和 OS/2 以及苹果的 Macintosh)上的不同应用程序之间转移数据,现在这种格式已经广泛用于 Windows 系统,因为它可以在 RichTextBox 控件中编辑。

Rtf 的版本从 1.0 到 1.9。Rtf 是 8 位格式的,为了方便传输,标准的 Rtf 文件只由 ASCII 字符组成,中文字符用转义符来表示。每个 Rtf 文件都是一个文本文件。文件开始处是{\rtf, 它作为 Rtf 文件的标志是必不可少的, Rtf 阅读器根据它来判断一个文件是否为 Rtf 格式。然后是文件头和正文,文件头包括字体表、文件表、颜色表等几个数据结构,正文中的字体、表格的风格就是根据文件头的信息来格式化的。每个表用一对大括号括起来,其中包含很多用字符“\”开始的命令。举个最简单的 Rtf 文件的例子,文件中只包含一行文字: {\rtflfoobar}。用写字板打开这个文件,显示“foobar”。

许多开源的 Rtf 文件解析器不能正确处理多字节编码内容,例如 javax.swing.text.rtf。当然有的项目也能够处理包含 Unicode 编码的 Rtf 文件,例如 RtfConverter(下载地址是

<http://www.codeproject.com/KB/recipes/RtfConverter.aspx>), 不过这个项目是 C#实现的, 后续我们将介绍如何用 Java 实现一个 Rtf 文件解析器 RtfConverter4J。

Rtf 文件解析器 RtfConverter4J 的设计目标是:

- 可以在各层次上分析 Rtf 数据。
- 把对 Rtf 数据的解析和解释分开。
- 保持解析器和解释器的可扩展性。
- Rtf 转换应用程序容易上手。
- 采用开放式架构, 能够很容易地自定义 Rtf 转换器。

因此, 设计了如图 3-14 所示的这个开放式架构。

RtfParser 类用于完成实际的数据解析。除了识别标签 (Tag), 也处理字符编码, 支持 Unicode。RtfParser 把 Rtf 数据分成如下几种基本的元素。

- RtfGroup: Rtf 元素的集合。
- RtfTag: 一个 Rtf 标签的名字和值对。
- RtfText: 任意的文本内容, 但是文本内容不一定是可见的。

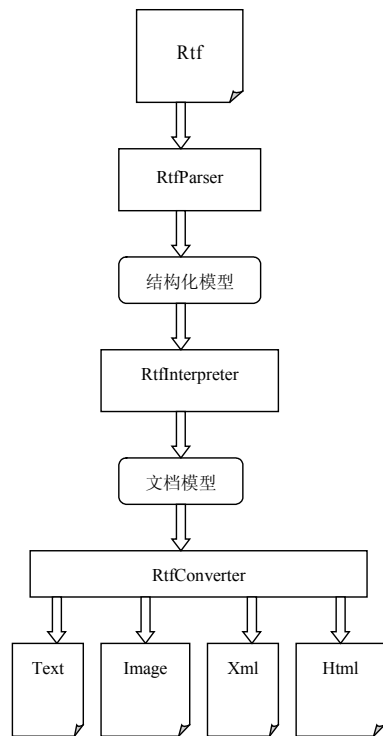


图 3-14 Rtf 转换程序总体结构图

在 RtfParser 中对 Unicode 编码的处理代码如下所示:

```
if (tagName.equals(RtfSpec.TagUnicodeCode)) {
    //读入 Unicode 编码的字符的值
    int unicodeValue = tag.getValueAsNumber();
    char unicodeChar = (char) unicodeValue;
    this.curText.append(unicodeChar);
    for (int i = 0; i < this.unicodeSkipCount; i++) {
        //跳过指定数量的文本
        char skip1 = (char) reader.read();
        if (skip1 == ' ') {
            char skip2 = (char) PeekNextChar(reader, false);
            if (skip2 == '\\') {
                reader.read();
                char skip3 = (char) PeekNextChar(reader, false);
                while (skip3 != '\\ ' && skip3 != '}' && skip3 != '{') {
```

```

        reader.read();
        skip3 = (char) PeekNextChar(reader, false);
    }
}
} else if (skip1 == '\\') {
    reader.read();
    char skip3 = (char) PeekNextChar(reader, false);
    while (skip3 != '\\\' && skip3 != '\'' && skip3 != '{') {
        reader.read();
        skip3 = (char) PeekNextChar(reader, false);
    }
} else if (skip1 == '\r') {
    reader.read();
    char skip2 = (char) PeekNextChar(reader, false);

    if (skip2 == '\\') {
        reader.read();
        char skip3 = (char) PeekNextChar(reader, false);
        while (skip3 != '\\\' && skip3 != '\'' && skip3 != '{') {
            reader.read();
            // System.Console.WriteLine((char) reader.Read());
            skip3 = (char) PeekNextChar(reader, false);
        }
    }
}
}
} else if (tagName.Equals(RtfSpec.TagUnicodeSkipCount)) {
    //读入 UnicodeSkipCount 的值
    int newSkipCount = tag.GetValueAsNumber();
    if (newSkipCount < 0 || newSkipCount > 10) {
        throw new Exception("invalid unicode skip count: " + tag);
    }
    this.unicodeSkipCount = newSkipCount;
}
}

```

因为 Rtf 数据中可能包含另外一种十六进制表示的 Unicode, 所以增加了对 TagUnicodeSkipCount 的处理。

Rtf 文件解析器的结构如图 3-15 所示。实际的解析过程可以通过 ParserListener 监听。ParserListener 通过观察者模式提供了对某个事件反应的机会并执行相应的动作。系统已经集成的解析器监听器 RtfParserListenerFileLogger 可以用来把 Rtf 元素的结构写入日志文件(主要用在开发阶段的调试)。输出可以通过 RtfParserLoggerSettings 进行配置。

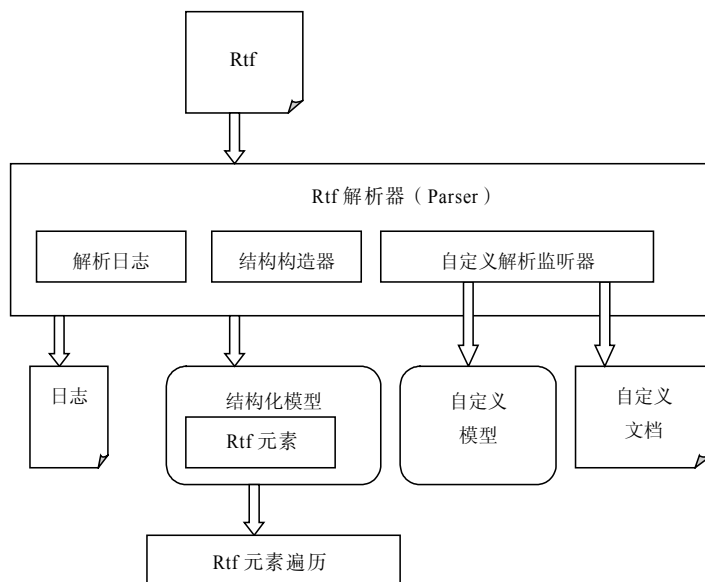


图 3-15 Rtf 解析器结构图

`RtfParserListenerStructureBuilder` 根据解析过程中碰到的 Rtf 元素生成结构化模型。这个模型把基本元素表示成 `IrtfGroup`、`IrtfTag` 和 `IrtfText` 的实例。可以通过 `RtfParserListenerStructureBuilder.StructureRoot` 取得层次结构。

当 Rtf 文档解析成结构化模型后，就可以通过 Rtf 解释器来解释。Rtf 文件解释器的结构如图 3-16 所示。解释结构化模型的一个方法是构造文档模型来提供对文档内容意义的高层次抽象。一个简单的文档模型由如下块组成——文档信息：标题、主题和作者等；用户属性；颜色信息；字体信息；文本格式；可视化信息。其中，可视化信息包括：

- 文本相关的格式化信息。
- 分隔符，如线、段落、节、页。
- 特殊字符，如制表符、段落开始/结束、下画线、空格、圆点、引号、连字符。
- 图片。

下面的例子展示了如何使用文档模型的高层 API：

```
//显示 Rtf 文件内容
public static void RtfWriteDocumentModel(String rtfStream) throws
Exception {
    RtfInterpreterListenerFileLogger logger = null;
    IRtfDocument document = RtfInterpreterTool.BuildDoc(rtfStream,
logger);
}
```

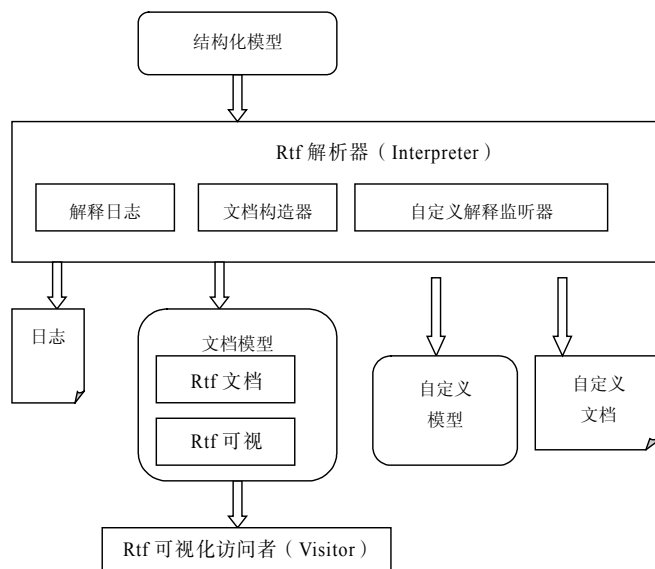


图 3-16 Rtf 解释器结构图

```

RtfWriteDocument(document);
} //RtfWriteDocumentModel

//根据文档模型遍历
public static void RtfWriteDocument(IRtfDocument document) {
    System.out.println("RTF Version: " + document.getRtfVersion());

    //显示文档信息
    System.out.println("Title: " + document.getDocumentInfo().getTitle());
    System.out.println("Subject: "
        + document.getDocumentInfo().getSubject());
    System.out.println("Author: " + document.getDocumentInfo().getAuthor());

    //显示字体表中的字体
    for (String fontName : document.getFontTable().keySet()) {
        System.out.println("Font: " + fontName);
    }

    //显示颜色表中的颜色
    for (IRtfColor color : document.getColorTable()) {
        System.out.println("Color: " + color.getAsDrawingColor());
    }

    //取得文档的用户属性，例如文档“创建者”或者“时间”
    for (IRtfDocumentProperty documentProperty : document

```





```
.getUserProperties()) {
    System.out.println("User property: " + documentProperty.getName());
}

//遍历所有可视化元素
for (IRtfVisual visual : document.getVisualContent()) {
    RtfVisualKind visualKind = visual.getKind();
    if (visualKind == RtfVisualKind.Text) { //文本
        System.out.println("Text: "
            + ((IRtfVisualText) visual).getText());
    } else if (visualKind == RtfVisualKind.Break) { //换行符号
        System.out.println("Tag: "
            + ((IRtfVisualBreak) visual).
            getBreakKind().toString());
    } else if (visualKind == RtfVisualKind.Special) { //特别字符
        System.out.println("Text: "
            + ((IRtfVisualSpecialChar) visual).getCharKind()
            .toString());
    } else if (visualKind == RtfVisualKind.Image) { //图像
        IRtfVisualImage image = (IRtfVisualImage) visual;
        System.out.println("Image: " + image.getFormat().toString()
            + " " + image.getWidth() + "x" + image.getHeight());
    }
}
}
```

下面调用 RtfConverter4J 提取文本：

```
URL u = new URL("http://www.silo.net/LaReja-2005-05-07/texto_chino_7M.rtf");
//创建一个 URL 连接对象
URLConnection uc = u.openConnection();
//提取内容并输出
InputStream is = uc.getInputStream();
RtfExtractor extractor=new RtfExtractor(is);
String text = extractor.getText();
is.close();
System.out.println("text:"+text);
```

提取 Rtf 文件的标题设计流程说明如下。

- ① 生成候选标题：把字体最大的文字块或者居中对齐的文字块作为候选标题。
- ② 评估候选标题：根据候选标题所在的位置和候选标题与正文的相似度来评分。

### ③ 输出标题模块：选择分值最大的标题输出。

首先定义候选标题类，代码如下所示：

```
public static class TitleInfo {
    public String fontName; //字体名
    public int fontSize; //字号
    public int position; //位置信息
    public int mergeTo; //合并块编号
    public boolean isBold; //是否粗体
    public String text; //内容
    public HashMap<String, Double> words; //内容切分结果
    public double weight; //权重
}
```

提取标题整体流程实现代码如下所示：

```
public static String getRtfTitle(String fileName) throws Exception {
    StringBuffer content = new StringBuffer(); //用来存储全文内容
    //取得候选标题
    ArrayList<TitleInfo> candidates=getCandidates(fileName,content);

    if (candidates == null || candidates.size() == 0) //没有候选标题
        return "";
    else if (candidates.size() == 1)
        return candidates.get(0).text;
    else {
        //候选标题评分
        rankTitle(candidates, content.toString());
        //把评分最高的候选标题作为标题提取结果
        return getBestTitle(candidates);
    }
}
```

取得候选标题部分实现代码如下所示：

```
public static ArrayList<TitleInfo> getCandidates(String fileName,
    StringBuffer content) {
    IRtfGroup rtfStructure = ParseRtf(fileName); //获取 Rtf 结构

    RtfInterpreterListenerDocumentBuilder docBuilder =
        new RtfInterpreterListenerDocumentBuilder(); //初始化
    RtfInterpreterListenerLogger interpreterLogger = null;
```



```
RtfInterpreterTool.Interpret(rtfStructure, interpreterLogger,
docBuilder);
RtfDocument doc = (RtfDocument) docBuilder.getDocument();

if (doc == null) //读取失败
    return null;
ArrayList<IRtfVisual> rtfVisuals = doc.getVisualContent();

int maxFontSize = 0;
int maxPosition = 0;
for (IRtfVisual rtfv : rtfVisuals) { //找最大字号
    if (rtfv.getKind() == RtfVisualKind.Text) {
        int currentFontSize = ((IRtfVisualText) rtfv).getFormat()
            .getFontSize();
        maxFontSize = Math.max(currentFontSize, maxFontSize);
    }
}

ArrayList<TitleInfo> candidates = new ArrayList<TitleInfo>();
for (int i = 0; i < rtfVisuals.size(); i++) {
    IRtfVisual rtfv = rtfVisuals.get(i);

    if (RtfVisualKind.Text == rtfv.getKind()) //只有 Text 才有文字属性
    {
        //换行前以第一种字符格式作为整行格式，除了字号。
        String fontName = ((IRtfVisualText) (rtfv)).getFormat()
            .getFont().getName(); // 首字符字体
        Color tc = ((IRtfVisualText) (rtfv)).getFormat()
            .getForegroundColor().getAsDrawingColor();
        boolean isBold = ((IRtfVisualText) (rtfv)).getFormat()
            .getIsBold();
        RtfTextAlignment alignment = ((IRtfVisualText) (rtfv))
            .getFormat().getAlignment();
        int fontSize = ((IRtfVisualText) (rtfv)).getFormat()
            .getFontSize();

        String oneRow = "";
        while (RtfVisualKind.Break != rtfv.getKind()) {
            //换行前的部分是一个整体
            if (rtfv.getKind() == RtfVisualKind.Text)
                oneRow += ((IRtfVisualText) rtfv).getText();
            i++;
            rtfv = rtfVisuals.get(i);
        }
    }
}
```

```

//公文属性判断
if (isProSendGovUnit(oneRow, tc) || //发文单位
    isProReceiGovUnit(oneRow, tc) || //收文单位
    isFileNum(oneRow)) { //文号
    maxPosition++;
    continue;
}

if (RtfTextAlignment.Center == alignment
    || fontSize == maxFontSize) {
    // 居中、最大字号加入候选标题
    candidates.add(
        new TitleInfo(oneRow,
            fontSize,
            maxPosition,
            fontName,
            isBold));
    } else
        //不居中的是正文内容
        content.append(oneRow + " ");
    }
    maxPosition++;
}

return candidates;
}

```

对候选标题评分的代码如下所示:

```

public static void rankTitle(ArrayList<TitleInfo> titles, String content){
    HashSet<String> stopWords = StopSet.getInstance(); //停用词表
    HashMap<String, Double> contentWords = new HashMap<String, Double>();

    int maxLength = 0;
    int maxFontSize = 0;
    int width = 440; //正文宽度
    int firstPosition = 9999;

    for (int i = 0, j = 1; j < titles.size(); i++, j++) {
        //第一步: 合并可能的标题
        TitleInfo ti = titles.get(i);
        TitleInfo tj = titles.get(j);
        firstPosition = Math.min(firstPosition, ti.position);
        if (ti.fontSize == tj.fontSize //字号大小一致
            && ti.position + 1 == tj.position //上下连贯

```



```
        && titles.get(i).text.length() > 1
        && titles.get(j).text.length() > 1) {
    if (!(tj.text.startsWith(" "))) {
        TitleInfo tTmp = ti;
        tj.mergeTo = i;
        while (tTmp.mergeTo != -1) {
            tj.mergeTo = tTmp.mergeTo;
            tTmp = titles.get(tTmp.mergeTo);
        }
        //向前合并标题同时删除;
        titles.get(tj.mergeTo).text = titles.get(tj.mergeTo).text
            .trim()
            + tj.text.trim();
        tj.text = " ";
        if (ti.fontSize * ti.text.length() > width) {
            //因为字符过多而换行的
            width *= 2;
            titles.get(tj.mergeTo).weight += 0.2;
        }
    }
}

for (TitleInfo m : titles) { //第二步：分词，确定候选标题的长度与位置范围
    if (m.text.trim().length() >= 2) { //非空标题分词
        maxLength = Math.max(maxLength, m.text.length());
        maxFontSize = Math.max(maxFontSize, m.fontSize);

        //标题分词，建向量
        ArrayList<CnToken> taggedTitle = Tagger.getFormatSegResult
            (m.text);

        m.words = new HashMap<String, Double>();
        for (CnToken ct : taggedTitle) {
            if ("m".equals(ct.type()) || "t".equals(ct.type())
                || stopWords.contains(ct.termText()))
                continue; //去除停用词

            Double val = m.words.get(ct.termText());
            if (val != null) {
                m.words.put(ct.termText(), new Double(val + 1.0));
            } else
                m.words.put(ct.termText(), new Double(1.0));
        }
    }
}
```

```

        m.position -= firstPosition;
    }
    //对正文分词，建向量
    ArrayList<CnToken> taggedContent = Tagger.getFormatSegResult(content);
    for (CnToken ct : taggedContent) {
        if ("w".equals(ct.type()) || "m".equals(ct.type())
            || "t".equals(ct.type())
            || stopWords.contains(ct.termText()))
            continue;//去除停用词

        if (contentWords.containsKey(ct.termText())) {
            contentWords.put(ct.termText(), new Double(contentWords.get(ct
                .termText()) + 1));
        } else
            contentWords.put(ct.termText(), new Double(1.0));
    }
    double contentNorm = calculateNorm(contentWords);

    for (int i = 0; i < titles.size(); i++){//第三步：综合评价候选标题权重
        TitleInfo t = titles.get(i);
        if (t.text.trim().length() >= 2) {
            double lengthWeight = getLengthWeight(t.text.length());
            double fontSizeWeight = getFontSizeWeight(t.fontSize,
                maxFontSize);
            double positionWeight = getPositionWeight(t.position);
            //计算可选标题与全文的相似度，用夹角余弦来衡量相似度
            double semanticWeight = getSimilarity(t.words, contentWords,
                contentNorm);

            //计算综合权重
            Double compositiveWeight = new Double(t.weight
                * Math.pow(lengthWeight * fontSizeWeight
                    * positionWeight, 1.0 / 3) * semanticWeight);
            //得出标题的最终权重
            if (t.isBold)
                t.weight = compositiveWeight * 1.1;
            else
                t.weight = compositiveWeight;
        }
    }
}

```

最后简单地取最大分值对应的标题：

```

public static String getBestTitle(ArrayList<TitleInfo> titles) {

```



```
double max = 0; //记录最大分值
String bestTitle = null; //记录最好标题
for (TitleInfo t : titles) {
    if (t.weight > max && t.text.trim().length() >= 2) {
        max = t.weight;
        bestTitle = t.text;
    }
}
return bestTitle;
}
```

### 3.2.5 Excel 文件

Excel 文件由一个工作簿（Workbook）组成。工作簿由一个或多个工作表（Sheet）组成，每个工作表都有自己的名称。每个工作表又包含多个单元格（Cell）。除了 POI 项目，还有开源项目 jxl (<http://www.andykhana.com/jexcelapi/index.html>) 可以用来读写 Excel 文件。

调用 Apache 的 POI 提取文本的代码如下所示：

```
public static String readDoc(InputStream is) throws IOException{
    //读入 Excel 文件数据流
    ExcelExtractor extractor = new ExcelExtractor(new POIFSFile
    System(is));
    //不返回公式
    extractor.setFormulasNotResults(true);
    //不包括 Sheet 名称
    extractor.setIncludeSheetNames(false);
    return extractor.getText();
}
```

为了提取 Excel 文件的标题，首先从每个工作表中找出最有可能的标题，然后从多个工作表中再次挑选最有可能的标题。

首先定义封装单元格属性的类，其中包含了用来计算标题重要度的一些属性：

```
public class CellInfo{
    public String text;//文本内容
    public short fontSize;//字号
    public short alignment;//对齐方式
    public boolean boldness;//是否黑体
    public int rowPos;//所在行的位置
    public double weight;//重要度
    public boolean isUnique;//独立成行
```

```

public CellInfo(String t, short fs, int rp, short align, boolean bold){
    text = t;
    fontSize = fs;
    rowPos = rp;
    alignment = align;
    boldness = bold;
    weight = 1.0;
    isUnique = false;
}
}

```

通过遍历工作表中的每个字符型单元格来取得每个工作表的最好标题:

```

private TitleInf getSheetBestTitle(HSSFSheet sheet, HSSFWorkbook wb){
    Iterator<HSSFRow> riter = sheet.rowIterator();//按行遍历工作表
    ArrayList<CellInfo> titles = new ArrayList<CellInfo>();

    int maxFontSize = 0;
    int rowCount = 0;
    while (riter.hasNext()){
        rowCount ++;
        int columnCount = 0;
        HSSFRow row = (HSSFRow) riter.next();//按行遍历
        Iterator<HSSFCell> citer = row.cellIterator();

        while(citer.hasNext()) {
            HSSFCell cell = citer.next();//每行再按列遍历
            int cellType = cell.getCellType();
            HSSFCellStyle cellStyle = cell.getCellStyle();

            if (cellType != HSSFCell.CELL_TYPE_BLANK ) { //非空
                columnCount ++;
            }
            if (cellType == HSSFCell.CELL_TYPE_STRING) { //字符型
                String cellString = cell.toString().trim();
                //取得单元格内的文本
                if (cellString.length() >= 2) {
                    HSSFFont cellFont = cell.getCellStyle().getFont(wb);

                    short fontheight = cellFont.getFontHeight();
                    short al = cellStyle.getAlignment();
                    short boldness = cellFont.getBoldweight() ;
                    maxFontSize = Math.max(maxFontSize, (int) fontheight);
                    CellInfo ci = new CellInfo(cellString,
                                                fontheight,
                                                rowCount,

```





```
        al,
        boldness == HSSFFont.BOLDWEIGHT_
        BOLD);
        titles.add(ci);
    }
}
}
if (columnCount == 1) //这行只有这个
    titles.get(titles.size() - 1).isUnique = true;
}

if (titles.size() == 0)
    return new TitleInf("", 0);
else
    return selectBestTitle(titles, maxFontSize, rowCount);
}
```

标题类包含了标题的文本内容和重要度：

```
public class TitleInf{
    public String text;//文本内容
    public double weight;//重要度
    public TitleInf(String t, double w){
        text = t;
        weight = w;
    }
}
```

取得整个 Excel 文件最有可能的标题：

```
public String getTitle(String fileName) throws Exception{
    InputStream is = new FileInputStream(fileName);
    HSSFWorkbook wb = new HSSFWorkbook(new POIFSFileSystem(is));
    ArrayList<TitleInf> candidate = new ArrayList<TitleInf>();//候选标题
    int activeSheetIndex = wb.getActiveSheetIndex();//取得活跃工作表的编号

    int sheetsNum = wb.getNumberOfSheets();
    for (int i = 0 ; i < sheetsNum ; i++){//取得每个工作表最有可能的标题
        TitleInf bti = getSheetBestTitle(wb.getSheetAt(i), wb);
        if (i == activeSheetIndex)
            bti.weight *= 3.0;
        candidate.add(bti);
    }

    //取得最评分最高的候选标题
    double maxWeight = 0;
```

```

String bestTitle = null;
for (TitleInf curTitle : candidate) {
    if (curTitle.weight >= maxWeight){
        maxWeight = curTitle.weight;
        bestTitle = curTitle.text;
    }
}

is.close();
return bestTitle;
}

```

### 3.2.6 PowerPoint 文件

在 PowerPoint 视图编辑区的上半部分显示幻灯片的缩略图，下半部分是备注编辑区。所以可提取的文本包括幻灯片显示的文本和备注中的文本。调用 Apache 的 POI 提取 PowerPoint 文件中的文本代码如下所示：

```

public static String readDoc(InputStream is) throws IOException{
    //创建 PowerPointExtractor
    PowerPointExtractor extractor=new PowerPointExtractor(is);
    //取得所有的幻灯片文本，但是不包括备注中的文本。
    //如果要返回备注中的文本，可以调用 setNotesByDefault(true)
    return extractor.getText();
}

```

PowerPoint 文件由一个或多个幻灯片（Slide）组成。第一张幻灯片的标题往往是整个 PowerPoint 文件的标题，提取标题实现代码如下所示：

```

SlideShow ss = new SlideShow(new HSLFSlideShow(is)); //is 是 ppt 文件的输入流
Slide[] slides = ss.getSlides(); //获得每一张幻灯片
return slides[0].getTitle(); //返回第一张幻灯片的标题

```



## 3.3 流媒体内容提取

相对于文本检索，音频和视频检索技术研究得比较少。常用的方法是提取出相关的文字描述来索引音频和视频。例如，把视频里面的声音通过语音识别（Speech Recognition），然后放入索引库，同时记录时间点。语音识别的方法有很多技术问题，比如电影里面有背景音乐，如何去除噪声和音乐比较麻烦，这将导致转换率低，而且多语言混杂，需要多种



语言处理包，另外，说话的人可能有各种方言，导致转换率非常低。另外把 rmvb 等视频文件格式中的字用屏幕文字识别转换也非常麻烦，因为涉及多语言的转换，尤其是汉字这样的大字符集文字，除非写得很标准。

视频搜索可以用在电台或电视台制作节目上，它们往往有录制了几十年的数据，现在想从中找出一些内容做合集，语音搜索和人脸搜索可以用上；视频网站每天都会有很多用户上传内容，这些有可能被插入一些违法的内容，如果全部人工审核的话，成本很大。先用一遍内容检索过滤，可以大大降低人工参与的工作量；用户个人拍录像时可以加一些语音标签，方便日后编辑。

### 3.3.1 音频流内容提取

Sphinx-4 (<http://cmusphinx.sourceforge.net/>) 是采用 Java 实现的一个语音识别软件。Sphinx 是一个基于隐马尔科夫模型的系统，首先它需要学习一套语音单元的特征，然后根据所学来推断出所需要识别的语音信号最可能的结果。学习语音单元特征的过程叫做训练。应用所学来识别语音的过程有时也被称为解码。在 Sphinx 系统中，训练部分由 Sphinx Trainer 来完成，解码部分由 Sphinx Decoder 来完成。为了识别普通话，可以使用 Sphinx Trainer 自己建立普通话的声学模型。训练时需要准备好语音信号（Acoustic Signals）与训练用语音信号对应的文本（Transcript File）。当前 Sphinx-4 只能使用 Sphinx-3 Trainer 生成的 Sphinx-3 声学模型。有计划创建 Sphinx-4 trainer 用来生成 Sphinx-4 专门的声学模型，但是这个工作还没完成。

讲稿（transcript）文件中记录了单词和非讲话声的序列。序列接着一个标记可以把这个序列和对应的语音信号关联起来。

例如有 160 个 wav 文件，每个文件对应一个句子的发音。例如，播放第一个声音文件，会听到 “a player threw the ball to me”，而且就这一句话。可以把这些 wav 或者 raw 格式的声音文件放到 myasm/wav 目录下。

接下来，需要一个控制文件。控制文件只是一个文本文件。这里把控制文件命名为 myam\_train.fields（必须把它命名成[name]\_train.fileids 的形式，这里[name]是任务的名字，例如 myam），其中有每个声音文件的名字（注意，没有文件扩展名）。

```
0001
0002
0003
0004
```

接下来，需要一个讲稿文件，文件中的每行有一个独立文件的发声，必须和控制文件相对应。例如，如果控制文件中第一行是 0001，因此讲稿文件中的第一行就是“A player threw the ball to me”，因为这是 0001.wav 的讲稿。讲稿文件也是一个文本文件，命名成 myam.corpus，应该有和控制文件同样多行。讲稿不包括标点符号，所以应删除所有标题符号，例如：

```
a player threw the ball to me
does he like to swim out to sea
how many fish are in the water
you are a good kind of person
```

以这样的顺序，对应 0001、0002、0003 和 0004 文件。

现在有了一些声音文件、一个控制文件和一个讲稿文件。

Sphinx-4 由 3 个主要模块组成：前端处理器（FrontEnd）、解码器（Decoder）和语言处理器（Linguist），其结构如图 3-17 所示。前端把一个或多个输入信号参数转化成特征序列。语言处理器把任何类型的标准语言模型和声学模型以及词典中的发声信息转换成为搜索图。这里，声学模型用来表示字符如何发音，语言模型用来评估一个句子的概率。解码器中的搜索管理器使用前端处理器生成的特征执行实际的解码并生成结果。在识别之前或识别过程中，应用程序都可以发出对每个模块的控制，这样就可以有效地参与到识别过程中来。

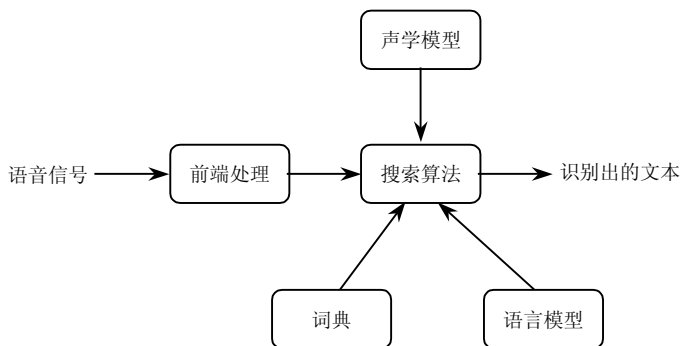


图 3-17 Sphinx-4 结构

语音识别的准确率受限于其识别的内容，内容越简单，则识别准确率越高，所以一般根据某个应用场景来识别语音。例如电视台要给录制的新闻节目加字幕，有批处理和实时翻译两种方式，这里采用批处理的方式。以识别新闻节目为例，开发流程说明如下。



- ① 准备新闻语料库：语料库就是一个文本文件，每行一个句子。
- ② 创建语言模型：一般采用基于统计的  $N$  元语言模型，例如 ARPA 格式的语言模型。可以使用语言模型工具 Kym (http://www.phontron.com/kym/) 生成 ARPA 格式的语言模型文件。
- ③ 创建发声词典：对于英文可以采用 ARPABET 格式注音的发音词典。由于汉语是由音节 (Syllable) 组成的语言，所以可以采用音节作为汉语语音识别基元。每个音节对应一个汉字，比较容易注音。此外，每个音节由声母和韵母组成，声韵母作为识别基元也是一种选择。
- ④ 设置配置文件：在配置文件中设置词典文件和语言模型路径。
- ⑤ 在 Eclipse 中执行语音识别的 Java 程序。初始情况下，需要执行 jsapi.exe 或 jsapi.sh 生成出 jsapi.jar 文件。

edu.cmu.sphinx.tools.feature.FeatureFileDumper 可以从音频文件中导出特征文件，例如 MFCC 特征。一般提取语音信号的频率特征，找出语音信号中的音节叫做端点检测，也就是找出每个字的开始端点和结束端点。因为语音信号中往往存在噪音，所以不是很容易找准端点。

Transcriber.jar 可以实现从声音文件中导出讲稿文件：

```
D:\sphinx4-1.0beta5\bin>java -jar -mx300M Transcriber.jar  
one zero zero zero one  
nine oh two one oh  
zero one eight zero three
```

### 3.3.2 视频流内容提取

视频信息一般由四部分组成：帧、镜头、情节、节目。视频流的内容可以从多个层次分析。

- 底层内容建模，包括颜色、纹理、形状、空间关系、运动信息等。
- 中层内容建模，指视频对象 (Video Object)，在 MPEG-4 中包括了视频对象。对象的划分可根据其独特的纹理、运动、形状、模型和高层语义为依据。
- 高层内容建模（语义概念等），例如一段体育视频，可以提取出扣篮或射门的片断。

关键帧提取策略说明如下。

- ① 设置一个最大关键帧数  $M$ 。
- ② 每个镜头的非边界过渡区的第一帧确定为关键帧。
- ③ 使用非极大值抑制法确定镜头边界系数极大值，并排序，以实现基于镜头边界系数的关键帧提取。

镜头边界检测方法有：只使用镜头边界系数的镜头边界检测——固定阈值法；结合相邻帧差的镜头边界检测——自适应阈值法。

可以使用 Java 媒体框架（JMF）API 在 Java 语言中处理声音和视频等时序性的媒体。下面我们首先通过 vid2jpg.java 类来提取视频中所有的帧。

```
/**
 * 用这个方法转换从 PushBufferDataSource 发出的数据
 */
public void transferData(PushBufferStream stream) {
    stream.read(readBuffer);

    //为了防止数据对象中的内容被其他线程修改
    Buffer inBuffer = (Buffer)(readBuffer.clone());

    //检查流是否已经结束
    if(readBuffer.isEOM()) {
        System.out.println("End of stream");
        return;
    }
    //实际把视频中的帧保存到文件
    useFrameData(inBuffer);
}
```

除了关键帧抽取，还可以考虑根据说话人识别和人脸识别来标注或搜索视频。例如，当某人说话的时候，找出视频库里所有该人说的话。对于有声音的视频可以利用视频中的音频信息检索内容。视频流和音频流在时间上是同步的。

手机应用 IntoNow 可以捕捉视频中的声音，通常它只需“听”20 秒左右，就可以判断出你正在看的是什么节目，并列出这个节目的基本信息，以及同样在看这个节目的其他 IntoNow 用户。



## 3.4 存储提取内容

可以把提取出的结果写到索引或者维基百科。例如，<http://jwbf.sourceforge.net/>可以实现把内容写到 MediaWiki：

```
MediaWikiBot mediaWikiBot = new MediaWikiBot(
    "http://wiki.1798hw.com/index.php");
// 维基百科首页
mediaWikiBot.login("1798hw", "1798hw"); // 登录到维基百科
String title = "黄山"; // 标题
Article article = new Article(mediaWikiBot, title); // 设置标题
article.setText("黄山内容介绍"); // 设置内容
article.save(); // 保存文章
```

为了实现自动写维基百科，可以把维基百科中在编辑状态的信息抓取过来：

```
http://zh.wikipedia.org/w/index.php?title=%E7%94%B5%E8%A7%86&action=edit
```

为了能够上传图片链接。需要修改 mediawiki 的配置文件 Localseting.php，增加以下内容：

```
$wgAllowExternalImages = true;
$wgAllowCopyUploads = true;
```

就可以上传外部图片链接地址。

在写入维基百科时，只需要加入图片地址，就会自动显示图片。以下是上传图片链接示例：

```
MediaWikiBot mediaWikiBot = new MediaWikiBot("http://wiki.1798hw.com/
index.php");
mediaWikiBot.login("1798hw", "1798hw");
Article article = new Article(mediaWikiBot, title);
article.setText(body);
article.addTextnl(imgUrl);
article.save();
```

但这样做图片仍然保存在远程服务器上，而不是本地地址，一般无法访问维基百科的图片，需要使用国外的代理来抓取维基百科中的正常图片。



### 3.5 本章小结

除了本章已经介绍过的 HTMLParser 和 NekoHTML 以外，Jsoup (<http://jsoup.org/>) 提供了类似于 jQuery 的操作方法来取出和操作数据。

如果用 Python，抓一个页面只需一句话：

```
html = urllib2.urlopen('http://www.baidu.com')
```

至于解析库，Python 中有基于 HTML 语法分析的 lxml.html、beautifulsope 和 PyQuery。Python 也有对正则表达式的支持。

本章介绍的从各种数据来源提取索引需要的信息，是爬虫开发中重要而且往往容易碰到问题的部分。各种文档格式处理方式总结如表 3-1 所示。

表 3-1 各种文档格式处理方式

格 式	解 析 包
Microsoft Office OLE2 Compound Document Format(Excel, Word, PowerPoint, Visio,Outlook)	Apache POI
Microsoft Office 2007 OOXML	Apache POI
Adobe Portable Document Format (PDF)	PDFBox
Rich Text Format (RTF)	RtfParser
英文文本	ICU4J
HTML	NekoHTML
XML	Java 的 javax.xml 类
ZIP Archives	Java 内部的 ZIP 类
TAR Archives	Apache Ant
GZIP compression	Java 内部的 GZIPInputStream
BZIP2 compression	Apache Ant
Image formats (metadata only)	Java 的 javax.imageio 类
Java class files	ASM library (JCR-1522)
Java JAR files	ZIP + Java Class files
MP3 audio	org.farng.mp3
Open Document	直接解析 XML
Microsoft Office 2007 XML	Apache POI
MIDI 文件	Java 内部的 javax.sound.midi.*
DWG 文件	DWGDirect





## 第 4 章

# 中文分词的原理与实现

有个经典笑话：护士看到病人在病房喝酒，就走过去小声叮嘱说：小心肝！病人微笑道：小宝贝。在这里，“小心肝！”这句话有歧义，从护士的角度理解是“小心/肝”，在病人的角度理解是“小心肝”。如果使用中文分词切分成“小心/肝”则可以消除这种歧义。

因为中文文本中词和词之间不像英文一样存在边界，所以中文分词是一个专业处理中文信息的搜索引擎首先面对的问题。英语、法语和德语等西方语言通常采用空格或标点符号将词隔开，具有天然的分隔符，所以词的获取比较简单。但是中文、日文和韩文等东方语言，虽然句子之间有分隔符，但词与词之间没有分隔符，所以需要靠程序切分出词。另外，除了可以用于全文查找，中文分词的方法也被应用到英语手写体识别中。因为在识别手写体时，单词之间的空格就不很清楚了。

要解决中文分词准确度的问题，是否提供一个免费版本的分词程序供人下载使用就够了？而像分词这样的自然语言处理领域的问题，很难彻底解决。例如，通用版本的分词也许需要做很多修改后才能用到手机上。所以需要让人能看懂其中的代码与实现原理，并参与到改进的过程中才能更好地应用。

本章的中文分词和下章介绍的文档排重和关键词提取等技术都属于自然语言处理技术的范围。因为在中文信息处理领域，中文分词一直是一个值得专门研究的问题，所以单独作为一章。



## 4.1 Lucene 中的中文分词

Lucene 中处理中文的常用方法有三种。以“咬死猎人的狗”这句话的输出结果为例。

- 单字方式：[咬] [死] [猎] [人] [的] [狗]。
- 二元覆盖的方式：[咬死] [死猎] [猎人] [人的] [的狗]。
- 分词的方式：[咬] [死] [猎人] [的] [狗]。

Lucene 中的 StandardTokenizer 采用了单字分词的方式。CJKTokenizer 采用了二元覆盖的实现方式。笔者开发的 CnTokenizer 采用了分词的方式，本章将介绍部分实现方法。

### 4.1.1 Lucene 切分原理

Lucene 中负责语言处理的部分在 org.apache.lucene.analysis 包。其中 TokenStream 类用来进行基本的分词工作，Analyzer 类是 TokenStream 的外围包装类，负责整个解析工作。有人把文本解析比喻成人体的消化过程，输入食物，分解出有用的氨基酸和葡萄糖等。Analyzer 类接收的是整段文本，解析出有意义的词语。

通常不需要直接调用分词的处理类 analysis，而是由 Lucene 内部来调用，其中：

- 在做索引阶段，调用 addDocument (doc) 时，Lucene 内部使用 Analyzer 来处理每个需要索引的列，如图 4-1 所示。

```
IndexWriter index = new IndexWriter(indexDirectory,
                                     new CnAnalyzer(), //用支持分词的分析器
                                     !incremental,
                                     IndexWriter.MaxFieldLength.UNLIMITED);
```

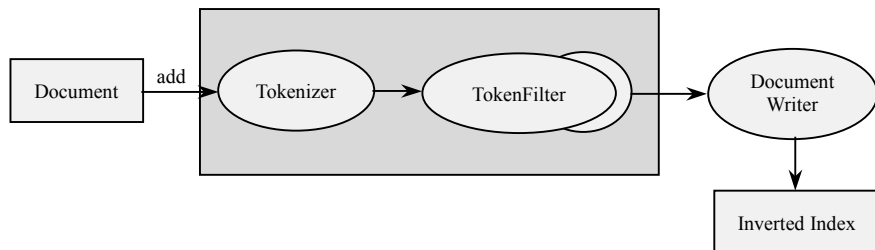


图 4-1 Lucene 对索引文本的处理

- 在搜索阶段，调用 `QueryParser.parse(queryText)` 来解析查询串时，`QueryParser` 会调用 `Analyzer` 来拆分查询字符串，但是对于通配符等查询不会调用 `Analyzer`。

```
Analyzer analyzer = new CnAnalyzer();           //支持中文的分词
QueryParser parser = new QueryParser(Version.LUCENE_CURRENT,"title",
analyzer);
```

因为在索引和搜索阶段都调用了分词过程，索引和搜索的切分处理要尽量一致，所以分词效果改变后需要重建索引。另外，可以用个速度快的版本，用来在搜索阶段切分用户的查询词，另外用一个准确切分的慢速版本用在索引阶段的分词。

为了测试 Lucene 的切分效果，下面是直接调用 `Analysis` 的例子：

```
Analyzer analyzer = new CnAnalyzer(); //创建一个中文分析器
//取得 Token 流
TokenStream ts = analyzer.tokenStream("myfield",new StringReader
("待切分文本"));
while (ts.incrementToken()) { //取得下一个词
    System.out.println("token: "+ts);
}
```

#### 4.1.2 Lucene 中的 Analyzer

为了更好地搜索中文，先通过图 4-2 了解一下在 Lucene 中通过 `WhitespaceTokenizer`、`WordDelimiterFilter`、`LowercaseFilter` 处理英文字符串的流程。

Lucene 中的 `StandardAnalyzer` 对于中文采用了单字切分的方式，这样的结果是单字匹配，如搜索“上海”，可能会返回和“海上”有关的结果。

`CJKAnalyzer` 采用了二元覆盖的方式实现。小规模搜索网站可以采用二元覆盖的方法，这样可以解决单字搜索“上海”和“海上”混淆的问题。采用中文分词的方法适用于中大规模的搜索引擎。猎兔搜索提供了一个基于 Lucene 接口的 Java 版中文分词系统。

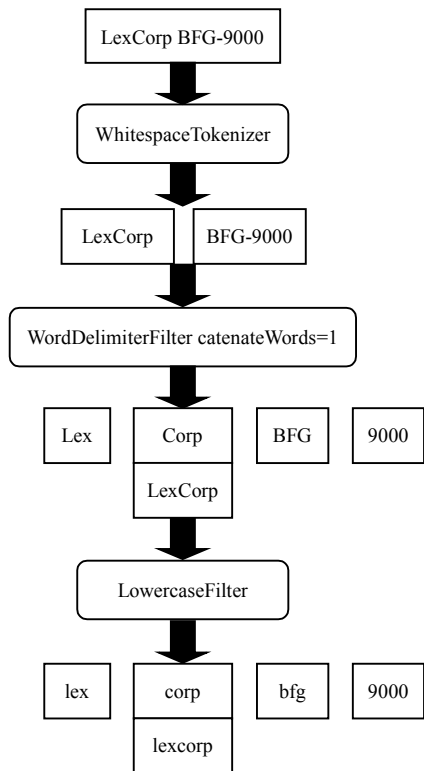


图 4-2 Lucene 处理英文字符串流程

可以对不同的索引列使用不同的 Analyzer 来切分。例如可以对公司名采用 CompanyAnalyzer 来分析；对地址列采用 AddressAnalyzer 来分析。这样可以通过更细分化的切分方式来实现更准确合适的切分效果。

例如把“唐山聚源食品有限公司”拆分成如表 4-1 所示的结果。

表 4-1 公司名拆分结果表

词	开始位置	结束位置	标注类型
唐山	0	2	City
聚源	2	4	KeyWord
食品	4	6	Feature
有限公司	6	10	Function

这里的开始位置和结束位置是指词在文本中的位置信息，也叫做偏移量。例如“唐山”这个词在“唐山聚源食品有限公司”中的位置是 0-2。OffsetAttribute 属性保存了词的位置信息；TypeAttribute 属性保存了词的类型信息。

切分公司名的流程如图 4-3 所示。

专门用来处理公司名的 CompanyAnalyzer 实现代码如下所示：

```
public class CompanyAnalyzer extends Analyzer {
    public TokenStream tokenStream
        (String fieldName, Reader reader) {
        //调用 ComTokenizer 切分公司名
        TokenStream stream = new
            ComTokenizer(reader);
        //调用 ComFilter 后续加工
        stream = new ComFilter
            (stream);
        return stream;
    }
}
```

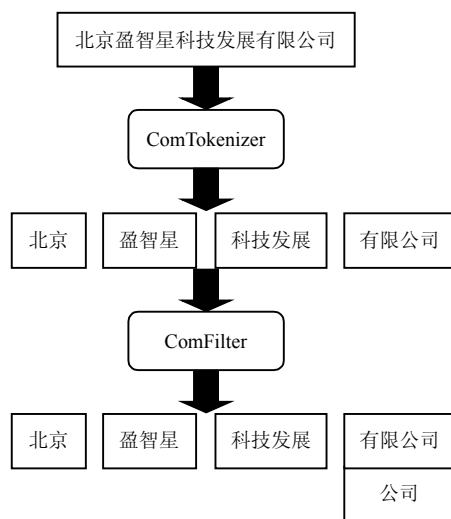


图 4-3 Lucene 处理公司名流程

对不同的数据写了不同用途的分析器，需要在同一个索引中针对不同的索引列使用。一般情况下只使用一个分析器。为了对不同的索引列使用不同的分析器，可以使用 PerFieldAnalyzerWrapper。在 PerFieldAnalyzerWrapper 中，可以指定一个默认的分析器，也



可以通过 `addAnalyzer` 方法对不同的列使用不同的分析器，例如：

```
PerFieldAnalyzerWrapper aWrapper =
    new PerFieldAnalyzerWrapper(new CnAnalyzer());
aWrapper.addAnalyzer("address", new AddAnalyzer());
aWrapper.addAnalyzer("companyName", new CompanyAnalyzer());
```

在这个例子中，将对所有的列使用 `CnAnalyzer`，除了地址列使用 `AddAnalyzer` 且公司名称列使用 `CompanyAnalyzer`。像其他分析器一样，`PerFieldAnalyzerWrapper` 可以在索引或者查询解析阶段使用。

### 4.1.3 自己写 Analyzer

一个简单的分析器（`Analyzer`）的例子如下所示：

```
public class MyAnalyzer extends Analyzer {
    public TokenStream tokenStream(String fieldName, Reader reader) {
        //以空格方式切分 Token
        TokenStream stream = new WhitespaceTokenizer(reader);
        //删除过短或过长的词，例如 in、of、it
        stream = new LengthFilter(stream, 3, Integer.MAX_VALUE);
        //给每个词标注词性
        stream = new PartOfSpeechAttributeImpl.PartOfSpeechTagging
            Filter(stream);
        return stream;
    }
}
```

一般在 `Tokenizer` 的子类实际执行词语的切分。需要设置的值有：和词相关的属性 `termAtt`、和位置相关的属性 `offsetAtt`。在搜索结果中高亮显示查询词时，需要用到和位置相关的属性。但是在切分用户查询词时，一般不需要和位置相关的属性。`Tokenizer` 的子类需要重写 `incrementToken` 方法。通过 `incrementToken` 方法遍历 `Tokenizer` 分析出的词，当还有词可以获取时，返回 `true`；已经遍历到结尾时，返回 `false`。

基于属性的方法把无用的词特征和想要的词特征分隔开。每个 `TokenStream` 在构造时增加它想要的属性。在 `TokenStream` 的整个生命周期中都保留一个属性的引用。这样在获取所有和 `TokenStream` 实例相关的属性时，可以保证属性的类型安全。

```
protected CnTokenStream(TokenStream input) {
    super(input);
    termAtt = (TermAttribute) addAttribute(TermAttribute.class);
}
```

在 `TokenStream.incrementToken()` 方法中，一个 `token` 流仅仅操作在构造方法中声明过的属性。例如，如果只要分词，则只需要 `TermAttribute`。其他的属性，例如 `PositionIncrementAttribute` 或者 `PayloadAttribute` 都被这个 `TokenStream` 忽略掉了，因为这时不需要其他的属性。

```
public boolean incrementToken() throws IOException {
    if (input.incrementToken()) {
        final char[] termBuffer = termAtt.termBuffer();
        final int termLength = termAtt.termLength();
        if (replaceChar(termBuffer, termLength)) {
            termAtt.setTermBuffer(output, 0, outputPos);
        }
        return true;
    }
    return false;
}
```

虽然也可以通过 `termAtt` 对象中的 `term` 方法返回词，但这个方法返回的是字符串，直接返回字符数组的 `termBuffer` 方法性能更好。下面是采用正向最大长度匹配实现的一个简单的 `Tokenizer`。

```
public class CnTokenizer extends Tokenizer {
    private static TernarySearchTrie dic = new TernarySearchTrie("SDIC.txt");
    //词典
    private TermAttribute termAtt; //词属性
    private static final int IO_BUFFER_SIZE = 4096;
    private char[] ioBuffer = new char[IO_BUFFER_SIZE];

    private boolean done;
    private int i = 0; //i 是用来控制匹配的起始位置的变量
    private int upto = 0;

    public CnTokenizer(Reader reader) {
        super(reader);
        this.termAtt = ((TermAttribute) addAttribute(TermAttribute.class));
        this.done = false;
    }

    public void resizeIOBuffer(int newSize) {
        if (ioBuffer.length < newSize) {
            //原数组不够大，创建一个新的更大的数组
            //原数组中的内容
            final char[] newCharBuffer = new char[newSize];
            System.arraycopy(ioBuffer, 0, newCharBuffer, 0, ioBuffer.

```



```
        length);
        ioBuffer = newCharBuffer;
    }
}

@Override
public boolean incrementToken() throws IOException {
    if (!done) {
        clearAttributes();
        done = true;
        upto = 0;
        i = 0;
        while (true) {
            final int length = input.read(ioBuffer, upto, ioBuffer.
                length
                    - upto);
            if (length == -1)
                break;
            upto += length;
            if (upto == ioBuffer.length)
                resizeIOBuffer(upto * 2);
        }

        if (i < upto) {
            char[] word = dic.matchLong(ioBuffer, i, upto);
            //正向最大长度匹配
            if (word != null) //已经匹配上
            {
                termAtt.setTermBuffer(word, 0, word.length);
                i += word.length;
            } else {
                termAtt.setTermBuffer(ioBuffer, i, 1);
                ++i; //下次匹配点在这个字符之后
            }
            return true;
        }
        return false;
    }
}
```

#### 4.1.4 Lietu 中文分词

Lietu 中文分词程序由 seg.jar 的程序包和一系列词典文件组成。通过系统参数 dic.dir 指定词典数据文件路径。我们可以写一个简单的分词测试代码：

```
String sentence = "有关刘晓庆偷税案";
//输入句子，返回单词组成的数组
String[] result = com.lietu(seg.result.Tagger.split(sentence);
for (int i=0; i<result.length;i++){
    System.out.println(result[i]);
}
```

使用分词的时候为了高亮显示关键字，必需保留词的位置信息。如果分词处理的网页有很多无意义的乱码，这些乱码导致的无意义的位置信息存储甚至可能会导致 5 倍以上的膨胀率。



## 4.2 查找词典算法

在讨论查找词典方法之前，首先看看词典格式。词典格式可以是方便人工查看和编辑的文本文件格式，也可以是方便机器读入的二进制格式，可以按约定存放在 `dic` 路径下或者由用户指定存放路径。

```
InputStream file = null;
if (System.getProperty("dic.dir")==null)//用户没有指定词典存放路径时，
//从默认的路径加载
    file = getClass().getResourceAsStream(Dictionary.getDir() + dic);
else
    file = new FileInputStream(new File(Dictionary.getDir() + dic));

BufferedReader in = new BufferedReader(new InputStreamReader(file, "GBK"));
String word;
while ((word = in.readLine()) != null) {
    //按行处理读入的文本格式的词典
}
in.close();
```

词典的最基本文本文件格式就是每行一个词。在基于词典的中文分词方法中，词典匹配算法是基础。使用的词典规模往往在几十万词以上。为了保证切分速度，需要选择一个好的查找词典算法。本节介绍词典的 Trie 树组织结构及词典的最大长度查找方法。

### 4.2.1 标准 Trie 树

散列是一种常见的高效查找方法，它根据数组下标查询，所以速度快。首先根据词表构造散列表，具体来说就是用给定的哈希函数构造词典到数组下标的映射，如果存在冲突，



则根据选择的冲突处理方法解决地址冲突。然后可以在哈希表的基础上执行哈希查找。

冲突导致散列性能降低。不存在冲突的散列表叫做完美散列（perfect hash）。整词散列不适合分词的最长匹配查找方式。

数字搜索树采用了逐字散列的方法，可以看成是一种逐字的完美散列。一个数字搜索 Trie 树（retrieve 树）的一个节点只保留一个字符。如果一个单词比一个字符长，则包含第一个字符的节点有指针指向下一个字符的节点，以此类推。这样组成一个层次结构的树，树的第一层包括所有单词的第一个字符，树的第二层包括所有单词的第二个字符，以此类推，数字搜索树的最大高度是词典中最长单词的长度。例如，如下单词序列组成的词典（as at be by he in is it of on or to）会生成如图 4-4 所示的数字搜索树：

数字搜索树的结构独立于生成数时单词进入的顺序。这里，Trie 树的高度是 2。因为树的高度很小，在数字搜索 Trie 树中搜索一个单词的速度很快。但是，这是以内存消耗为代价的，树中的每一个节点都需要很多内存。假设每个词都是由 26 个小写英文字母中的一个组成的，这个节点中会有 26 个指针。所以不太可能直接用这样的数字搜索树来存储中文这样的大字符集。

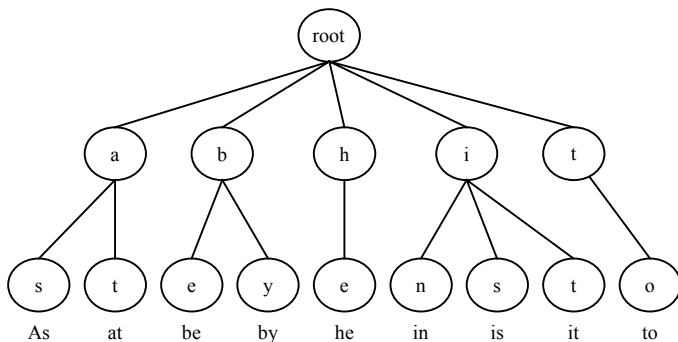


图 4-4 数字搜索树

Trie 树在实现上有一个树类（SearchTrie）和一个节点类（TrieNode）。SearchTrie 的主要方法有两个：

- 增加单词到搜索树，方法原型是：addWord（String word）。
- 从文本的指定位置开始匹配单词，方法原型是：matchLong（String text,int offset）。

给定一个固定电话号码，找出这个电话号码对应的区域。固定电话号码都是以 0 开始的多位数字，可以通过给定电话号码的前缀找出对应的地区，例如：

0995:新疆:托克逊县  
 0856:贵州:铜仁  
 0996:新疆:焉耆回族自治县

可以使用数字搜索树算法快速查找电话号码前缀。  
 例如: 0371 这个区号有四个节点对应, 也就是说有 4 个 TrieNode 对象。第一个对象中的 splitChar 属性是 0; 第二个对象中的 splitChar 属性是 3; 第三个对象中的 splitChar 属性是 7; 第四个对象中的 splitChar 属性是 1。每个 TrieNode 对象有 10 个孩子节点, 分别对应孩子节点的 splitChar 为 0 到 9。0371 形成的 Trie 树如图 4-5 所示。

Trie 树的节点类定义如下:

```
public static final class TrieNode {
    protected TrieNode[] children; //孩子节点
    protected char splitChar; //分隔字符
    protected String area; //电话所属地区信息

    /**
     * 构造方法
     *
     * @param splitchar 分隔字符
     */
    protected TrieNode(char splitchar) {
        children = new TrieNode[10];
        area = null;
        this.splitChar = splitchar;
    }
}
```

加载词, 形成数字搜索树的方法如下:

```
private void addWord(String string, TSTNode root, String area) {
    TSTNode tstNode = root;
    for (int i = 1; i < string.length(); i++) {
        char c0 = string.charAt(i);
        int ind = Integer.parseInt(string.substring(i, i + 1));
        TSTNode tmpNode = tstNode.children[ind];
        if (null == tmpNode) {
            tmpNode = new TSTNode(c0);
        }
    }
}
```

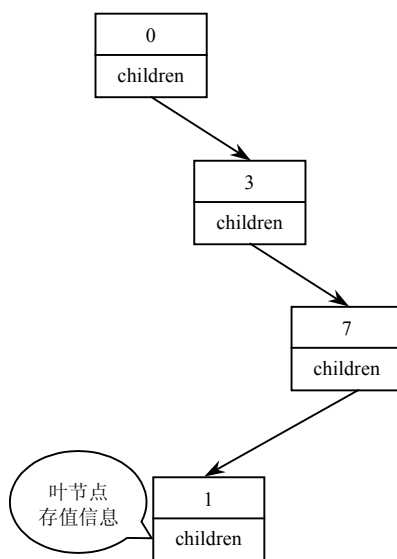


图 4-5 电话号码组成的数字搜索树



```
    }  
    if (i == string.length() - 1) {  
        tmpNode.area = area;  
    }  
    tstNode.children[ind] = tmpNode;  
    tstNode = tmpNode;  
}  
}
```

查询的过程对于查询词来说，从前往后一个字符一个字符的匹配。对于 Trie 树来说，是从根节点往下匹配的过程。从给定电话号码搜索前缀的方法如下所示：

```
public String search(String tel) {  
    TrieNode tstNode = root;  
    for (int i = 1; i < tel.length(); i++) { //从前往后一个字符一个字符的匹配  
        tstNode = tstNode.children[(tel.charAt(i) - '0')];  
        if (null != tstNode.area) {  
            return tstNode.area;  
        }  
    }  
    return null; //没找到  
}
```

考虑把 Trie 树改成通用的结构，使用散列表存储孩子节点，并使用范型定义值类型。

```
public class TrieNode<T> {  
    private Character splitChar; //分隔字符  
    private T nodeValue; //值信息  
    private Map<Character, TrieNode<T>> children =  
        new HashMap<Character, TrieNode<T>>();  
    //孩子节点  
}
```

## 4.2.2 三叉 Trie 树

在一个三叉搜索树（Ternary Search Trie）中，每一个节点包括一个字符，但和数字搜索树不同，三叉搜索树只有三个指针：一个指向左边的树；一个指向右边的树；还有一个向下，指向单词的下一个数据单元。三叉搜索树是二叉搜索树和数字搜索树的混合体。它有和数字搜索树差不多的速度但是和二叉搜索树一样只需要相对较少的内存空间。

树是否平衡取决于单词的读入顺序。如果按排序后的顺序插入，则生成方式最不平衡。单词的读入顺序对于创建平衡的三叉搜索树很重要，但对于二叉搜索树就不太重要。通过

选择一个排序后数据单元集合的中间值，并把它作为开始节点，我们可以创建一个平衡的三叉树。可以写一个专门的过程来生成平衡的三叉树词典。

```
/**
 * 在调用此方法前，先把词典数组 k 排好序
 * @param fp 写入的平衡序的词典
 * @param k 排好序的词典数组
 * @param offset 偏移量
 * @param n 长度
 * @throws Exception
 */
void outputBalanced(BufferedWriter fp, ArrayList<String> k, int offset,
int n){
    int m;
    if (n < 1) {
        return;
    }
    m = n >> 1; //m=n/2

    String item= k.get(m + offset);

    fp.write(item); //把词条写入到文件
    fp.write('\n');

    outputBalanced(fp, k, offset, m); //输出左半部分
    outputBalanced(fp, k, offset + m + 1, n - m - 1); //输出右半部分
}
```

取得平衡的单词排序类似于洗扑克牌。假想有若干张扑克牌，每张牌对应一个单词，先把牌排好序，然后取最中间的一张牌，单独放着。剩下的牌分成了两摞，左边一摞牌中也取最中间的一张放在取出来的那张牌后面。右边一摞牌中也取最中间的一张放在取出来的牌后面，以此类推。

我们再次以有序的数据单元 (as at be by he in is it of on or to) 为例。首先我们把关键字 “is” 作为中间值并且构建一个包含字母 “i” 的根节点。它的直接后继节点包含字母 “s” 并且可以存储任何与 “is” 有关联的数据。对于 “i” 的左树，我们选择 “be” 作为中间值并且创建一个包含字母 “b” 的节点，字母 “b” 的直接后继节点包含 “e”。该数据存储在 “e” 节点。对于 “i” 的右树，按照逻辑，选择 “on” 作为中间值，并且创建 “o” 节点以及它的直接后继节点 “n”。最终的三叉树如图 4-6 所示。

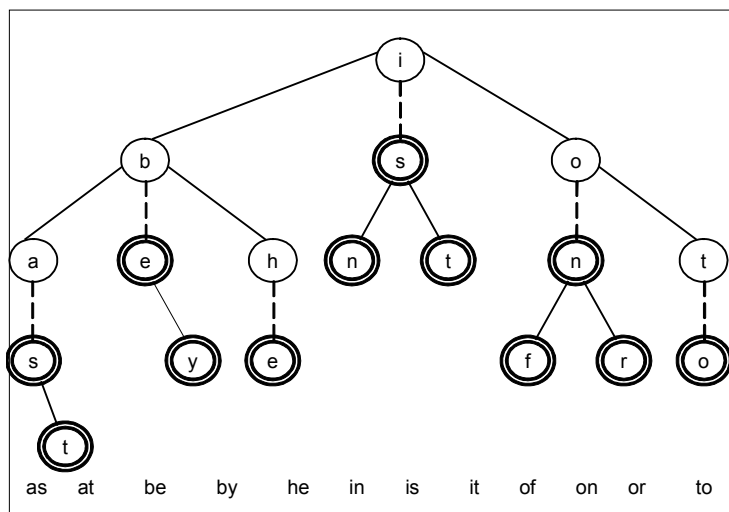


图 4-6 三叉树

垂直的虚线代表一个父节点下面的直接后继节点。只有父节点和它的直接后继节点才能形成一个数据单元的关键字；“i”和“s”形成关键字“is”，但是“i”和“b”不能形成关键字，因为它们之间仅用一条斜线相连，不具有直接后继关系。图 4-6 中带圈的节点为终止节点，如果查找一个词以终止节点结束，则说明三叉树包含这个词。从根节点开始查找单词。以搜索单词“is”为例，向下到相等的孩子节点“s”，在两次比较后找到“is”。查找“ax”时，执行三次比较达到首字符“a”，然后经过两次比较到达第二个字符“x”，返回结果是“ax”不在树中。

TernarySearchTrie 本身存储关键字到值的对应关系，可以当作 HashMap 对象来使用。关键字按照字符拆分成许多节点，以 TSTNode 的实例存在。值存储在 TSTNode 的 data 属性中。TSTNode 的实现代码如下所示：

```
public final class TSTNode {
    /** 节点的值 */
    public Data data=null;//data 属性可以存储词原文和词性、词频等相关的信息

    protected TSTNode loNode; //左边节点
    protected TSTNode eqNode; //中间节点
    protected TSTNode hiNode; //右边节点

    protected char splitchar; //本节点表示的字符
    /**
     * 构造方法
     */
}
```

```

    *@param splitchar 该节点表示的字符
    */
    protected TSTNode(char splitchar) {
        this.splitchar = splitchar;
    }
    public String toString() {
        return "splitchar:" + splitchar;
    }
}

```

查找词典的基本过程是：输入一个词，返回这个词对应的 TSTNode 对象，如果该词不在词典中则返回空。在查找词典的过程中，从树的根节点匹配 Key，按 Char 从前往后匹配 Key。charIndex 表示 Key 当前要比较的 Char 的位置。匹配过程如下所示：

```

protected TSTNode getNode(String key, TSTNode startNode) {
    if (key == null ) {
        return null;
    }
    int len = key.length();
    if (len ==0)
        return null;
    TSTNode currentNode = startNode; //匹配过程中当前节点的位置
    int charIndex = 0;
    char cmpChar = key.charAt(charIndex);
    int charComp;
    while (true) {
        if (currentNode == null) { //没找到
            return null;
        }
        charComp = cmpChar - currentNode.splitchar;
        if (charComp == 0) { //相等
            charIndex++;
            if (charIndex == len) { //找到了
                return currentNode;
            }
        }
        else {
            cmpChar = key.charAt(charIndex);
        }
        currentNode = currentNode.eqNode;
    }
    else if (charComp < 0) { //小于
        currentNode = currentNode.loNode;
    }
    else { //大于
        currentNode = currentNode.hiNode;
    }
}

```



```
    }  
}
```

三叉树的创建过程也就是在 Trie 树上创建和单词对应的节点。实现代码如下所示：

```
//向词典树中加入一个单词的过程  
private TSTNode addWord(String key) {  
    TSTNode currentNode = root; //从树的根节点开始查找  
    int charIndex = 0; //从词的开头匹配  
    while (true) {  
        //比较词的当前字符与节点的当前字符  
        int charComp = key.charAt(charIndex) - currentNode.splitchar;  
        if (charComp == 0) { //相等  
            charIndex++;  
            if (charIndex == key.length()) {  
                return currentNode;  
            }  
            if (currentNode.eqNode == null) {  
                currentNode.eqNode = new TSTNode(key.charAt(  
                    charIndex));  
            }  
            currentNode = currentNode.eqNode;  
        } else if (charComp < 0) { //小于  
            if (currentNode.loNode == null) {  
                currentNode.loNode = new TSTNode(key.charAt(  
                    charIndex));  
            }  
            currentNode = currentNode.loNode;  
        } else { //大于  
            if (currentNode.hiNode == null) {  
                currentNode.hiNode = new TSTNode(key.charAt(  
                    charIndex));  
            }  
            currentNode = currentNode.hiNode;  
        }  
    }  
}
```

相对于查找过程，创建过程在搜索过程中判断出链接的空值后创建相关的节点，而不是碰到空值后结束搜索过程并返回空值。

同一个词可以有不同的词性，例如“朝阳”既可能是一个“区”，也可能是一个“市”。可以把这些和某个词的词性相关的信息放在同一个链表中。这个链表可以存储在 TSTNode 的 Data 属性中。



### 4.3 中文分词的原理

中文分词就是对中文断句，这样能消除文字的部分歧义。除了基本的分词功能，为了消除歧义还可以进行更多的加工。中文分词可以分成如下几个子任务。

- 分词：把输入的标题或者文本内容等分成词。
- 词性标注 (POS)：给分出来的词标注上名词或动词等词性。词性标注可以部分消除词的歧义，例如“行”作为量词和作为形容词表示的意思不一样。
- 语义标注：把每个词标注上语义编码。

很多分词方法都借助词库。词库的来源是语料库或者词典，例如“人民日报语料库”或者《现代汉语大词典》。

为了探索适合中国国情的中文分词，不妨借鉴一下汽车产业曾经发生过的事情。日本的汽车充电电池公司使用全自动化生产线，大部分工作都交给机器人来完成，这样的生产线建一条至少需要数千万元的投资，可位于深圳的比亚迪公司把生产线分解成一个个可以人工完成的工序，在最后容易产生误差的环节则设计了很多简单实用的夹具来保证质量。因为加工后的语料库往往很稀缺，为了使得人工可调整分词结果，这里所有的词典都采用方便人工编辑的文本格式。

中文分词有以下两类方法。

- 机械匹配的方法：例如正向最大长度匹配 (Forward Maximum Match) 的方法和逆向最大长度匹配 (Reverse Maximum Matching) 的方法。
- 统计的方法：例如概率语言模型分词方法和最大熵的分词方法等。

正向最大长度匹配的分词方法实现起来很简单。每次从词典中查找和待匹配串前缀最长匹配的词，如果找到匹配词，则把这个词作为切分词，待匹配串减去该词；如果词典中没有词与其匹配，则按单字切分。例如，Trie 树结构的词典中包括如下的词语：

大 大学 大学生 活动 生活 中 中心 心

为了形成平衡的 Trie 树，把词先排序，结果为：

中 中心 大 大学 大学生 心 活动 生活



按平衡方式生成的词典 Trie 树如图 4-7 所示，其中粗黑显示的节点可以作为匹配终止节点。

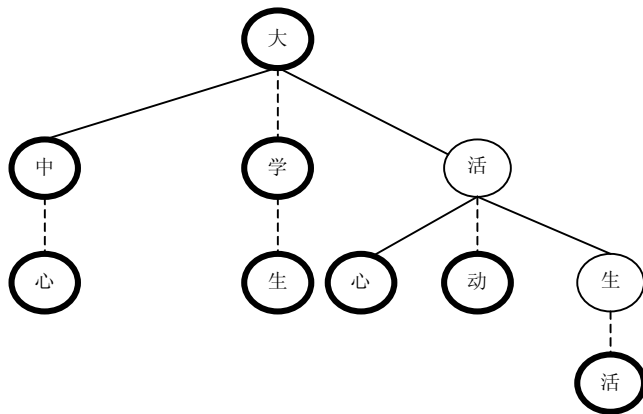


图 4-7 三叉树

输入“大学生活动中心”，首先匹配出“大学生”，然后匹配出“活动”，最后匹配出“中心”。切分过程如表 4-2 所示。

表 4-2 正向最大长度匹配切分过程表

已匹配上的结果	待匹配串
NULL	大学生活动中心
大学生	活动中心
大学生/活动	中心
大学生/活动/中心	NULL

最后分词结果为“大学生/活动/中心”。

在最大长度匹配的分词方法中，需要用到从指定字符串返回指定位置的最长匹配词的方法。例如，当输入串是“大学生活动中心”，则返回“大学生”这个词，而不是返回“大”或者“大学”。从 Trie 树搜索最长匹配单词的方法如下所示：

```

public String matchLong(String key,int offset) { //输入字符串和匹配的起始位置
    String ret = null;
    if (key == null || rootNode == null || "".equals(key)) {
        return ret;
    }
    TSTNode currentNode = rootNode;
    int charIndex = offset;
    while (true) {

```

```

        if (currentNode == null) {
            return ret;
        }
        int charComp = key.charAt(charIndex) - currentNode.splitter;

        if (charComp == 0) {
            charIndex++;

            if (currentNode.data != null) {
                ret = currentNode.data; //候选最长匹配词
            }
            if (charIndex == key.length()) {
                return ret; //已经匹配完
            }
            currentNode = currentNode.eqNode;
        } else if (charComp < 0) {
            currentNode = currentNode.loNode;
        } else {
            currentNode = currentNode.hiNode;
        }
    }
}

```

测试 `matchLong` 方法如下所示:

```

String sentence = "大学生活动中心";//输入字符串
int offset = 0;//匹配的开始位置
String ret = dic.matchLong(sentence, offset);
System.out.println(sentence+" match:"+ret);

```

返回结果如下所示:

大学生活动中心 match:大学生

正向最大长度分词的实现代码如下所示:

```

public void wordSegment(String sentence) { //传入一个字符串作为要处理的对象
    int senLen = sentence.length(); //首先计算出传入的字符串的字符长度
    int i=0; //控制匹配的起始位置

    while(i < senLen) { //如果 i 小于此字符串的长度就继续匹配
        String word = dic.matchLong(sentence, i); //正向最大长度匹配
        if(word!=null) { //已经匹配上
            //下次匹配点在这个词之后
        }
    }
}

```

```

        i += word.length();
        //如果这个词是词库中的那么就打印出来
        System.out.print(word + " ");
    }
    else{//如果在词典中没有找到匹配上的词，就按单字切分
        word = sentence.substring(i, i+1);
        //打印一个字
        System.out.print(word + " ");
        ++i;//下次匹配点在这个字符之后
    }
}
}
}

```

因为采用了 Trie 树结构查找单词，所以和用 HashMap 查找单词的方式比较起来，这种实现方法代码更简单，而且切分速度更快。例如“有意见分歧”这句话，正向最大长度切分的结果是“有意/见/分歧”，逆向最大长度切分的结果是“有/意见/分歧”。因为汉语的主干成分后置，所以逆向最大长度切分的精确度稍高。另外一种最少切分的方法是使每一句中切出的词数最小。



## 4.4 中文分词流程与结构

中文分词总体流程与结构如图 4-8 所示。

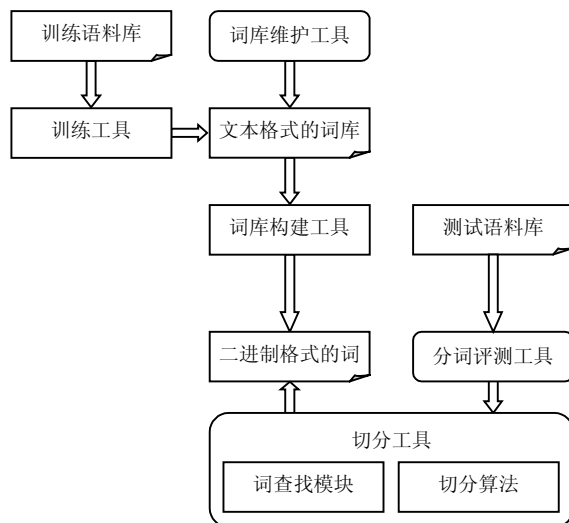


图 4-8 中文分词结构图

简化版本的中文分词切分过程说明如下。

- ① 生成全切分词图：根据基本词库对句子进行全切分，并且生成一个邻接链表表示的词图。
- ② 计算最佳切分路径：在这个词图的基础上，运用动态规划算法生成切分最佳路径。
- ③ 词性标注：可以采用 HMM 方法进行词性标注。
- ④ 未登录词识别：应用规则识别未登录词。
- ⑤ 按需要的格式输出结果。

复杂版本的中文分词切分过程说明如下。

- ① 对输入字符串切分成句子：对一段文本进行切分，依次从这段文本中切分出一个句子，然后对这个句子再进行切分。
- ② 原子切分：对于一个句子的切分，首先是通过原子切分，将整个句子切分成一个个的原子单元（即不可再切分的形式，例如 ATM 这样的英文单词可以看成不可再切分的）。
- ③ 生成全切分词图：根据基本词库对句子进行全切分，并且生成一个邻接链表表示的词图。
- ④ 计算最佳切分路径：在这个词图的基础上，运用动态规划算法生成切分最佳路径。
- ⑤ 未登录词识别：进行中国人名、外国人名、地名、机构名等未登录名词的识别。
- ⑥ 重新计算最佳切分路径。
- ⑦ 词性标注：可以采用 HMM 方法或最大熵方法等进行词性标注。
- ⑧ 根据规则调整切分结果：根据每个分词的词形以及词性进行简单的规则处理，如日期分词的合并。
- ⑨ 按需要的格式输出结果：例如输出成 Lucene 需要的格式。



## 4.5 形成切分词图

为了消除分词中的歧异，提高切分准确度，需要找出一句话所有可能的词，生成全切分词图。

如果待切分的字符串有  $m$  个字符，考虑每个字符左边和右边的位置，则有  $m+1$  个点对应，点的编号从 0 到  $m$ 。把候选词看成边，可以根据词典生成一个切分词图。切分词图是一个有向正权重的图。“有意见分歧”这句话的切分词图如图 4-9 所示。

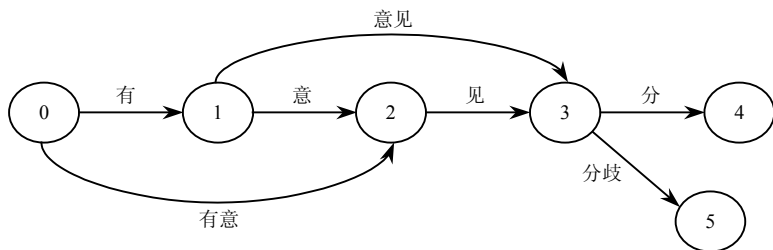


图 4-9 中文分词切分路径

在“有意见分歧”的切分词图中：“有”这条边的起点是 0，终点是 1；“意见”这条边的起点是 0，终点是 2，以此类推。切分方案就是从源点 0 到终点 5 之间的路径，共存在两条切分路径。

- 路径 1：0—1—3—5 对应切分方案  $S_1$ ：有/ 意见/ 分歧/
- 路径 2：0—2—3—5 对应切分方案  $S_2$ ：有意/ 见/ 分歧/

切分词图中的边都是词典中的词，边的起点和终点分别是词的开始和结束位置。

```

public class CnToken{
    public String termText;//词
    public int start;//词的开始位置
    public int end;//词的结束位置
    public int freq;//词在语料库中出现的频率
    public CnToken(int vertexFrom, int vertexTo, String word) {
        start = vertexFrom;
        end = vertexTo;
        termText = word;
    }
}

```

邻接表表示的切分词图由一个链表表示的数组组成。首先实现一个单向链表：

```
public class TokenLinkedList implements Iterable<TokenInf> {
    public static class Node {
        public TokenInf item;
        public Node next;

        Node(TokenInf item) {
            this.item = item;
            next = null;
        }
    }

    private Node head;

    public TokenLinkedList() {
        head = null;
    }

    public void put(TokenInf item) {
        Node n = new Node(item);
        n.next = head;
        head = n;
    }

    public Node getHead() {
        return head;
    }

    public Iterator<TokenInf> iterator() { //迭代器
        return new LinkIterator(head);
    }

    private class LinkIterator implements Iterator<TokenInf> {
        Node itr;

        public LinkIterator(Node begin) {
            itr = begin;
        }

        public boolean hasNext() {
            return itr != null;
        }

        public TokenInf next() {
```



```
        if (itr == null) {
            throw new NoSuchElementException();
        }
        Node cur = itr;
        itr = itr.next;
        return cur.item;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}

public String toString() {
    StringBuilder buf = new StringBuilder();
    Node cur = head;

    while (pCur != null) {
        buf.append(cur.item.toString());
        buf.append('\t');
        cur = cur.next;
    }

    return buf.toString();
}
}
```

为了方便用动态规划的方法计算最佳前驱词，在此单向链表的基础上形成逆向邻接表：

```
public class AdjList {
    private TokenLinkedList list[]; //AdjList 的图结构

    /**
     * 构造方法：分配空间
     */
    public AdjList(int verticesNum) {
        list = new TokenLinkedList[verticesNum];

        //初始化数组中所有的链表
        for (int index = 0; index < verticesNum; index++) {
            list[index] = new TokenLinkedList();
        }
    }

    public int getVerticesNum() {
```

```

        return list.length;
    }

    /**
     * 增加一个边到图中
     */
    public void addEdge(TokenInf newEdge) {
        list[newEdge.end].put(newEdge);
    }

    /**
     * 返回一个迭代器，包含以指定点结尾的所有的边
     */
    public Iterator<TokenInf> getAdjacencies(int vertex) {
        TokenLinkedList ll = list[vertex];
        if(ll == null)
            return null;
        return ll.iterator();
    }
}

```

首先从词典中形成以某个字符串的前缀开始的词集合。例如，以“中华人民共和国成立了”这个字符串前缀开始的词集合是“中”、“中华”、“中华人民共和国”，一共三个词。这三个词都存在于当前的词典中。如果要找出指定位置开始的所有词，下面是匹配的方法：

```

public static class PrefixRet {
    public ArrayList<String> values;
    public int end; //记录下次匹配的起始位置
}

//如果匹配上则返回 true，否则返回 false
public boolean getMatch(String sentence, int offset, PrefixRet prefix){
    if (sentence == null || rootNode == null || "".equals(sentence)) {
        return false;
    }
    boolean match = matchEnglish(offset, sentence, prefix);
    if (match) {
        return true;
    }

    match = matchNum(offset, sentence, prefix);
    if (match) {
        return true;
    }
}

```





```
prefix.end = offset+1;
ArrayList<String> ret = new ArrayList<String>();
TSTNode currentNode = rootNode;
int charIndex = offset;
while (true) {
    if (currentNode == null) {
        prefix.values = ret;
        if (ret.size() > 0) {
            return true;
        }
        return false;
    }
    int charComp = sentence.charAt(charIndex) - currentNode.splitChar;
    if (charComp == 0) {
        charIndex++;
        if (currentNode.data != null) {
            //System.out.println(currentNode.data) ;
            ret.add(currentNode.data);
        }
        if (charIndex == sentence.length()) {
            prefix.values = ret;
            if (ret.size() > 0) {
                return true;
            }
            return false;
        }
        currentNode = currentNode.eqNode;
    } else if (charComp < 0) {
        currentNode = currentNode.loNode;
    } else {
        currentNode = currentNode.hiNode;
    }
}
}
```

对于英文和数字等需要特殊处理。“ATM 柜员机”这个字符串前缀开始的词集合是“ATM”。这里的“ATM”并不一定存在于当前的词典中。为了区分这两种情况，可以把每类需要特殊处理的词放在单独的过程中执行。

```
//匹配英文的过程。返回第一个不是英文的位置
int matchEnglish (int start, String sentence)
//匹配数字的过程。返回第一个不是数字的位置
int matchNum (int start, String sentence)
```

然后在统一的过程中调用 `matchEnglish` 和 `matchNum`。

```
getMatch(String sentence, int offset){
    int ret= matchEnglish(offset,key);
    if(ret>offset)
return EnglishWord;
    ret = matchNum(offset,key);
    if(ret>offset)
return NumWord;
    //匹配普通词
}
```

这里还要用到对字符类型的判断，具体包括字母、数字、中文。

```
public enum CharType {
    /** char type: SINGLE byte */
    SINGLE,
    /** char type: delimiter */
    DELIMITER,
    /** char type: Chinese Char */
    CHINESE,
    /** char type: letter */
    LETTER,
    /** char type: chinese number */
    NUM,
    /** char type: index */
    INDEX,
    /** char type: other */
    OTHER
}
```

可以用查表法来快速判断字符类型。

通过查词典形成切分词图的主体过程如下所示：

```
for(int i=0;i<len;){
    boolean match = dict.getMatch(sentence, i, wordMatch);//到词典中查询
    if (match) { //已经匹配上
        for (String word:wordMatch.values) { //把查询到的词作为边加入切分词图中
            j = i+word.length();
            g.addEdge(new CnToken(i, j, 10, word));
        }
        i=wordMatch.end;
    }else{//把单字作为边加入切分词图中
```



对于函数  $y=\log(x)$ ，当  $x$  增大时， $y$  也会增大，所以是单调递增函数。取  $\log$  后，表示一个小于 1 的正数的精确度加大了。

$$P(S) \approx P(w_1) \times P(w_2) \times \cdots \times P(w_m) \propto \log P(w_1) + \log P(w_2) + \cdots + \log P(w_m)$$

这里的  $\propto$  是正比符号。因为词的概率小于 1，所以取  $\log$  后是负数。最后算  $\log P(w)$ 。

其中，对于不同的  $S$ ， $m$  的值是不一样的，一般来说  $m$  越大， $P(S)$  会越小。也就是说，分出的词越多，概率越小。这符合实际的观察，如最大长度匹配切分往往会使得  $m$  较小。

这个计算  $P(S)$  的公式也叫做基于一元概率语言模型的计算公式。这种分词方法简称一元分词。它综合考虑了切分出的词数和词频。一般来说，词数少、词频高的切分方案概率更高。考虑一种特殊的情况：所有词的出现概率相同，则一元分词退化成最少词切分方法。

计算任意一个词出现的概率的方法如下：

$$P(W_i) = \frac{W_i \text{在语料库中的出现次数 } n}{\text{语料库中的总词数 } N}$$

因此：

$$\log P(W_i) = \log(\text{Freq}_w) - \log N$$

如果词概率的对数值事前已经算出来了，则结果直接用加法就可以得到  $\log P(S)$ ，而加法比乘法速度更快。

从另外一个角度来看，计算最大概率等于求切分词图的最短路径。但是这里不采用 Dijkstra 算法，而采用动态规划的方法求解最短路径。

常用的词语概率表如表 4-3 所示。

$$P(S_1) = P(\text{有}) \times P(\text{意见}) \times P(\text{分歧}) = 1.8 \times 10^{-9}$$

$$P(S_2) = P(\text{有意}) \times P(\text{见}) \times P(\text{分歧}) = 1 \times 10^{-11}$$

可得  $P(S_1) > P(S_2)$ ，所以选择  $S_1$  对应的切分。

如何尽快找到概率最大的词串？因为假设每个词之间的概率是上下文无关的，因此满足用动态规划求解所要求的最优子结构性质和无后效性。

表 4-3 词语概率表

词 语	概 率
...	...
有	0.0180
有意	0.0005
意见	0.0010
见	0.0002
分歧	0.0001
...	...



在动态规划求解的过程中并没有先生成所有可能的切分路径  $S_i$ ，而是求出值最大的  $P(S_i)$  后，利用回溯的方法直接输出  $S_i$ 。

到节点  $\text{Node}_i$  为止的最大概率称为节点  $\text{Node}_i$  的概率，因此：

$$P(\text{Node}_i) = P_{\max}(W_1, W_2, \dots, W_i) = \max_{W_j \in \text{prev}(\text{Node}_i)} (P(\text{StartNode}(W_j))) * P(W_j)$$

如果  $W_j$  的结束节点是  $\text{Node}_i$ ，就称  $W_j$  为  $\text{Node}_i$  的前驱词。这里的  $\text{prev}(\text{Node}_i)$  就是节点  $i$  的前驱词集合。

比如上面的例子中，候选词“有”就是节点 1 的前驱词，“意见”和“见”都是节点 3 的前驱词。

$\text{StartNode}(w_j)$  是  $w_j$  的开始节点，也是节点  $i$  的前驱节点。

因此切分的最大概率  $\max(P(S))$  就是：

$$P(\text{Node}_m) = P(\text{节点 } m \text{ 的最佳前驱节点}) \times P(\text{节点 } m \text{ 的最佳前驱词})$$

按节点编号，从前往后计算如下：

$$P(\text{Node}_0) = 1$$

$$P(\text{Node}_1) = P(\text{有})$$

$$P(\text{Node}_3) = P(\text{Node}_1) \times P(\text{意见})$$

修改词的描述类如下所示：

```
public class CnToken {
    public String termText; //词
    public String type; //词性
    public int start; //开始位置
    public int end; //结束位置
    public double logProb; //概率取对数
}
```

首先把切分方案抽象成一个邻接矩阵表示的图，然后从左到右计算最佳前驱节点。

//计算节点 i 的最佳前驱节点，以及它的概率

```

void getPrev(AdjList g,int i){
    //得到前驱词的集合
    Iterator<CnToken> it = g.getPrev(i);
    double maxProb = Double.NEGATIVE_INFINITY;
    int maxNode = -1;
    //根据前驱词集合挑选最佳前趋节点
    while(it.hasNext()) {
        CnToken itr =it.next();
        double nodeProb = prob[itr.start]+itr.logProb;//候选节点概率
        if (nodeProb > maxProb) {
            //候选节点概率最大的开始节点就是最佳前趋节点
            maxNode = itr.start;
            maxProb = nodeProb;
        }
    }
    prob[i] = maxProb;//节点概率
    prevNode[i] = maxNode;//最佳前驱节点
}

```

最后从终节点回溯找出最大概率的切分路径:

```

ArrayList<Integer> ret = new ArrayList<Integer>();
//从右向左取最佳前驱节点，通过回溯发现最佳切分路径
for(int i=(g.verticesNum-1); i>0;i=prevNode[i]) {
    ret.add(i);
}

```

上面的计算中假设相邻两个词之间是上下文无关的，但实际情况并不如此，例如：前面一个词是数词，后面一个词更有可能是量词。如果前后两个词都只有一种词性，则可以利用词之间的搭配信息对分词决策提供帮助。

```

while (it.hasNext()) {
    CnToken itr = it.next();
    long currentCost = itr.cost;
    //前一个词到下一个词的转移概率
    int transProb = getTransProb(itr.type, i.type);
    currentCost += transProb;//注意不要让 currentCost 的值溢出

    if (currentCost > maxFee) {
        maxID = itr;
        maxFee = currentCost;
    }
}

```



## 4.7 $N$ 元分词方法

为了切分更准确，要考虑一个词所处的上下文。例如“上海银行间的拆借利率上升”，因为“银行”后面出现了“间”这个词，所以把“上海银行”分成“上海”和“银行”两个词。

一元分词假设前后两个词的出现概率是相互独立的，但实际上不太可能。比如，“沙县小吃”附近经常有“桂林米粉”，所以这两个词是正相关。但是很少有人把“沙县小吃”和“星巴克”相提并论。[羡慕][嫉妒][恨]这三个词有时候会连续出现。切分出来的词序列越通顺，越有可能是正确的切分方案。 $N$ 元模型主要用来衡量词序列搭配的合理性。

估计单词  $w_1$  后出现  $w_2$  的概率。根据条件概率的定义：

$$P(w_2 | w_1) = \frac{P(w_1, w_2)}{P(w_1)}$$

可以得到：

$$P(w_1, w_2) = P(w_1)P(w_2 | w_1)$$

同理：

$$P(w_1, w_2, w_3) = P(w_1, w_2)P(w_3 | w_1, w_2)$$

所以有：

$$P(w_1, w_2, w_3) = P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2)$$

更加一般的形式：

$$P(S) = P(w_1, w_2, \dots, w_n) = P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2) \cdots P(w_n | w_1 w_2 \cdots w_{n-1})$$

这叫做概率的链规则。其中， $P(w_2 | w_1)$  表示  $w_1$  之后出现  $w_2$  的概率。如果词  $w_1$  和  $w_2$  独立出现，则  $P(w_2 | w_1)$  等价于  $P(w_2)$ 。

这样需要考虑在  $n-1$  个单词序列后出现的单词  $w$  的概率。直接使用这个公式计算  $P(S)$  存在两个致命缺陷：一个缺陷是参数空间过大，不可能实用化；另外一个缺陷是数据稀疏严重。例如，词汇量( $V$ ) = 20,000 时，可能的二元(bigrams)组合数量有 400 000 000 个。可

能的三元(trigrams) 组合数量有 8 000 000 000 000 个。可能的四元(4-grams)组合数量有  $1.6 \times 10^{17}$  个。为了解决这两个问题，我们引入了马尔科夫假设：一个词的出现仅仅依赖于它前面出现的有限的一个或者几个词。

如果简化成一个词的出现仅仅依赖于它前面出现的一个词，那么就称为二元模型，即：

$$P(S) = P(w_1, w_2, \dots, w_n) = P(w_1) P(w_2|w_1) P(w_3|w_1, w_2) \dots P(w_n|w_1 w_2 \dots w_{n-1})$$

$$\approx P(w_1) P(w_2|w_1) P(w_3|w_2) \dots P(w_n|w_{n-1})$$

如果简化成一个词的出现仅仅依赖于它前面出现的两个词，就称为三元模型。

如果切分方案  $S$  是  $n$  个词组成的，那么  $P(w_1) P(w_2|w_1) P(w_3|w_2) \dots P(w_n|w_{n-1})$  也是  $n$  项连乘积。无论采用一元模型还是二元模型或者三元模型都是  $n$  项连乘积。只不过二元以上模型是条件概率的连乘积。例如，对于切分“有意见分歧”来说，二元模型计算：

$$P(\text{有}) P(\text{意见}|\text{有}) P(\text{分歧}|\text{意见})$$

三元模型计算：

$$P(\text{有}) P(\text{意见}|\text{有}) P(\text{分歧}|\text{有, 意见})$$

在实践中用得最多的就是二元模型和三元模型了，而且效果很不错。高于四元的模型用得很少，因为训练它需要更庞大的语料，而且数据稀疏严重，时间复杂度高，精度却提高得不多。

二元模型考虑一个单词后出现另外一个单词的概率，是  $N$  元模型中的一种。例如：一般来说，“中国”之后出现“北京”的概率大于“中国”之后出现“北海”的概率，也就是：

$$P(\text{北京}|\text{中国}) > P(\text{北海}|\text{中国})$$

$$P(w_i|w_{i-1}) \approx \text{freq}(w_{i-1}, w_i) / \text{freq}(w_{i-1})$$

二元词表的格式是“左词@右词:组合频率”，例如：

```
中国@北京:100
中国@北海:1
```

可以把二元词表看成是基本词表的常用搭配。分词初始化时，先加载基本词表，对每



个词编号，然后加载二元词表，只存储词的编号。

对于拼音转换等歧义较多的情况也可以采用三元模型（Trigram），例如：

$$P(\text{海淀}|\text{中国,北京}) > P(\text{海龙}|\text{中国,北京})$$

$$P(w_i|w_{i-2},w_{i-1}) \approx \text{freq}(w_{i-2},w_{i-1},w_i) / \text{freq}(w_{i-2},w_{i-1})$$

因为有些词作为开始词的可能性比较大，例如“在那遥远的地方”、“在很久以前”，这两个短语都以“在”这个词作为开始词。因此，在实际的  $N$  元分词过程中，增加虚拟的开始节点（Start）和结束节点（End），分词过程中考虑  $P(\text{在}|\text{Start})$ 。如果把“有意见分歧”当成一个完整的输入，分词结果实际是：“Start/ 有/ 意见/ 分歧/ End”。

下面我们来实现二元分词。把二元分词看成是二阶马尔可夫过程，计算  $P(\text{见}|\text{意})$ ，看成是节点组合  $\{1,2\}$  转移到节点组合  $\{2,3\}$  的概率，写成  $P(\{1,2\} \rightarrow \{2,3\})$ 。可以想象成在下跳棋，跳两次涉及 3 个位置，二元连接中的前后两个词涉及 3 个节点。

一个词的开始位置和结束位置组成的节点组合是二元词图中的点。前后两个词的转移概率作为边的权重。“有意见分歧”这句话中节点的组合有： $\{0,1\}$ 、 $\{0,2\}$ 、 $\{1,2\}$ 、 $\{1,3\}$ 、 $\{2,3\}$ 、 $\{3,4\}$ 、 $\{3,5\}$ 。得到的二元切分词图如图 4-10 所示。

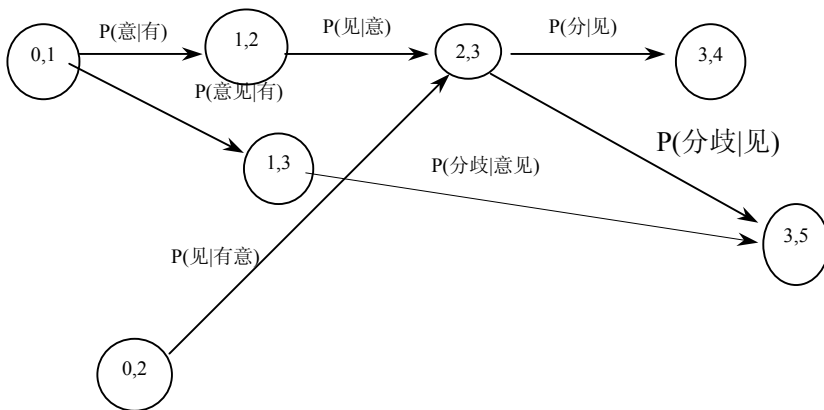


图 4-10 二元词图

这个二元切分词图可以看成是以词为基础的，如图 4-11 所示。

用动态规划的方法求解二元切分词图最短路径的伪代码如下：

```
for(currentWord : segGraph){ //从前往后遍历切分词图中的每个词
```

```

//得到当前词的前驱词集合
WordArray prevWordList = segGraph.prevWordList(currentWord.start);
double wordProb = Double.MAX_VALUE; //候选词概率
for( prevWord : prevWordList ){
    double currentProb = transProb(prevWord,currentWord) + prevWord.prob;
    if(transProb<minProb){
        wordProb = currentProb;
        minNode = nextWords;
    }
}
currentWord.bestPrev = minNode; //设置当前词的最佳前驱词
currentWord.prob = wordProb; //设置当前词的概率
}

```

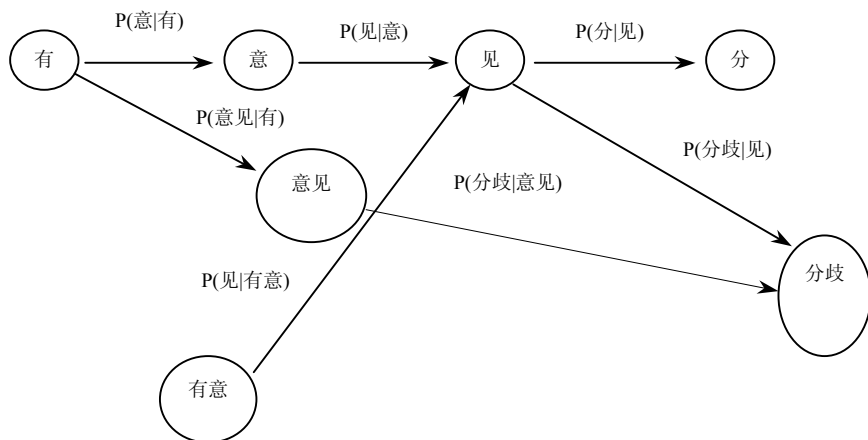


图 4-11 词表示的二元切分词图

一元分词一个节点，二元分词两个节点组合， $n$  元分词  $n$  个节点组合。如果把词序列的概率看成马尔可夫过程：一元分词看成是一阶马尔可夫过程，计算  $P(\text{见})$  看成是节点 2 转移到节点 3 的概率，写成  $P(2 \rightarrow 3)$ ；二元分词看成是二阶马尔可夫过程，计算  $P(\text{见}|\text{意})$ ，看成是节点组合  $\{1,2\}$  转移到节点组合  $\{2,3\}$  的概率，写成  $P(\{1,2\} \rightarrow 3)$ ；三元分词看成是三阶马尔可夫过程，计算  $P(\text{见}|\text{有},\text{意})$ ，看成是节点组合  $\{0,1,2\}$  转移到节点组合  $\{1,2,3\}$  的概率，写成  $P(\{0,1,2\} \rightarrow 3)$ 。所有有效的 2 节点组合组成二元词图中的节点，所有有效的 3 节点组合组成三元词图中的节点，依此类推，所有有效的  $n$  节点组合组成  $n$  元词图中的节点。

查找  $N$  元词典的方法有：可以采用 Trie 树的形式来存放  $N$  元模型的参数。与词典 Trie 树的区别在于：词典 Trie 树上每个节点对应一个汉字，而  $N$  元模型 Trie 树的一个节点对应一个词。或者可以把搭配信息存放在词典 Trie 树的叶子节点上。存储从词编号到频率的映射，采用折半查找。



```
public class BigramMap {
    public int[] keys;//词编号
    public int[] vals;//频率
}
```

在自然语言处理中， $N$  元模型可以应用于字符，衡量字符之间的搭配；或者应用于词，衡量词之间的搭配。可以应用于编码识别，将要识别的文本按照 GB 码和 BIG5 码分别识别成不同的汉字串，然后计算其中所有汉字频率的乘积，取乘积大的一种编码。



## 4.8 新词发现

词典中没有的，但是结合紧密的字或词有可能组成一个新词。比如：“水立方”如果不在词典中，可能会切分成两个词“水”和“立方”。如果在一篇文档中“水”和“立方”结合紧密，则“水立方”可能是一个新词。可以用信息熵来度量两个词的结合紧密程度。信息熵的一般公式是：

$$I(X, Y) = \log_2 \frac{P(X, Y)}{P(X)P(Y)}$$

如果  $X$  和  $Y$  的出现相互独立，则  $P(X, Y)$  的值和  $p(X)p(Y)$  的值相等， $I(X, Y)$  为 0。如果  $X$  和  $Y$  密切相关， $P(X, Y)$  将比  $P(X)P(Y)$  大很多， $I(X, Y)$  值也就远大于 0。如果  $X$  和  $Y$  的几乎不会相邻出现，而它们各自出现的概率又比较大，那么  $I(X, Y)$  将取负值，这时候  $X$  和  $Y$  负相关。设  $f(C)$  是词  $C$  出现的次数， $N$  是文档的总词数，则：

$$P(C_1, C_2) = P(C_1) * P(C_2 | C_1) = \frac{f(C_1)}{N} * \frac{f(C_1 C_2)}{f(C_1)} = \frac{f(C_1 C_2)}{N}$$

因此，两个词的信息熵计算如下：

$$I(C_1, C_2) = \log_2 N + \log_2 \frac{f(C_1 C_2)}{f(C_1)f(C_2)} = \log_2 N + \log_2 f(C_1 C_2) - \log_2 f(C_1) - \log_2 f(C_2)$$

两个相邻出现的词叫做二元连接串。

```
public class Bigram {
    String one;//上一个词
    String two;//下一个词
}
```

```

private int hashvalue = 0;

    Bigram(String first, String second) {
        this.one = first;
        this.two = second;
        this.hashvalue = (one.hashCode() ^ two.hashCode());
    }
}

```

从二元连接串中计算其信息熵的代码如下所示：

```

int index = 0;
fullResults = new BigramsCounts[table.size()];
Bigrams key;
int freq; //频率
double logn = Math.log((double)n); //文档的总词数取对数
double temp;
double entropy;
int bigramCount; //f(c1,c2)
for( Entry<Bigrams,int[]> e : table.entrySet()){//计算每个二元连接串的信息熵
    key = e.getKey();
    freq1 = oneFreq.get(key.one).freq;
    freq2 = oneFreq.get(key.two).freq;
    temp = Math.log((double)freq1) + Math.log((double)freq2);
    bigramCount = (e.getValue())[0];
    entropy = logn+Math.log((double)bigramCount) - temp; //信息熵
    fullResults[index++] =
        new BigramsCounts(
            bigramCount,
            entropy,
            key.one,
            key.two);
}

```

新词语有些具有普遍意义的构词规则，例如“模仿秀”由“动词+名词”组成。统计的方法和规则的方法结合对每个文档中重复子串组成的候选新词打分，超过阈值的候选新词选定为新词。此外，可以用 Web 信息挖掘的方法辅助发现新词：网页锚点上的文字可能是新词，例如“美甲”。另外，可以考虑先对文档集合聚类，然后从聚合出来的相关文档中挖掘新词。



## 4.9 未登录词识别

“南京市长江大桥？”“你怎么知道的？”“因为看到一个标语——南京市长江大桥欢迎您。”

在分词时即时发现词表中没有的词叫做未登录词识别。未登录词在英文中叫做 Out Of Vocabulary（简称 OOV）词。常见的未登录词包括人名、地名、机构名。

对识别未登录人名有用的信息说明如下。

- 未登录词所在的上下文。例如：“\*\*教授”，这里“教授”是人名的下文；“邀请\*\*”，这里“邀请”是人名的上文。
- 未登录词本身的概率。例如：不依赖上下文，直观地来看，“刘宇”可能是个人名，“史光”不太可能是个人名。采用未登录词的概率作为这种可能性的衡量依据。“刘宇”作为人名的概率等于“刘宇”作为人名出现的次数除以人名出现的总次数。

例如：我爸叫李刚。这里“动词 + 姓 + 名 + 标点符号”组成了一个识别规则。可以根据这个识别规则识别出“李刚”这个人名。这个规则的完整形式是：

动词 + 中国人名 + 标点符号 => 动词 + 姓 + 名 + 标点符号

所以可以通过匹配规则来识别未登录词。为了实现同时查找多个规则，可以把右边的模式组织成 Trie 树，左边的模式作为节点属性。全切分词图匹配上右边的模式后用左边的模式替换。

可以用二元模型或三元模型来整合未登录词本身的概率和未登录词所在的上下文这两种信息。

未登录地名识别过程说明如下。

- ① 选取未登录地名候选串。
- ② 未登录地名特征识别。

③ 对每个候选未登录地名根据特征判断是否真的地名。判断方法可以用 SVM 二值分类。

④ 整合地名词图。



## 4.10 词性标注

词性用来描述一个词在上下文中的作用。例如描述一个概念的词叫做名词，在下文引用这个名词的词叫做代词。有的词性经常会出现一些新的词，例如名词，这样的词性叫做开放式词性。另外一些词性中的词比较固定，例如代词，这样的词性叫做封闭式词性。因为存在一个词对应多个词性的现象，所以给词准确地标注词性并不是很容易。比如：“改革”在“中国开始对计划经济体制进行**改革**”这句话中是一个动词，在“医药卫生**改革**中的经济问题”中是一个名词。把这个问题抽象出来就是已知单词序列  $W_1 W_2 L W_n$ ，给每个单词标注上词性  $C_1 C_2 L C_n$ 。

不同的语言有不同的词性标注集。比如英文有反身代词，例如 **myself**，而中文则没有反身代词。为了方便指明词的词性，可以给每个词性编码。例如《PFR 人民日报标注语料库》中把“形容词”编码成 **a**；名词编码成 **n**；动词编码成 **v** 等。

词性标注有小标注集和大标注集。例如小标注集把代词都归为一类，大标注集可以把代词进一步分成三类。

- 人称代词：你 我 他 它 你们 我们 他们
- 疑问代词：哪里 什么 怎么
- 指示代词：这里 那里 这些 那些

采用小标注集比较容易实现，但是太小的标注集可能会导致类型区分度不够。例如在黑白色世界中，可以通过颜色的深浅来分辨出物体，但是通过七彩颜色可以分辨出更多的物体。

参考《PFR 人民日报标注语料库》的词性编码表，如表 4-4 所示。



表 4-4 词性编码表

代 码	名 称	举 例
a	形容词	最/d 大/a 的/u
ad	副形词	一定/d 能够/v 顺利/ad 实现/v 。/w
ag	形语素	喜/v 煞/ag 人/n
an	名形词	人民/n 的/u 根本/a 利益/n 和/c 国家/n 的/u 安稳/an 。/w
B	区别词	副/b 书记/n 王/nr 思齐/nr
c	连词	全军/n 和/c 武警/n 先进/a 典型/n 代表/n
d	副词	两侧/f 台柱/n 上/f 分别/d 雄踞/v 着/u
dg	副语素	用/v 不/d 甚/dg 流利/a 的/u 中文/nz 主持/v 节目/n 。/w
e	叹词	啊/e ！/w
f	方位词	从/p 一/m 大/a 堆/q 档案/n 中/f 发现/v 了/u
g	语素	例如 dg 或 ag
h	前接成分	目前/t 各种/r 非/h 合作制/n 的/u 农产品/n
i	成语	提高/v 农民/n 讨价还价/i 的/u 能力/n 。/w
j	简称略语	民主/ad 选举/v 村委会/j 的/u 工作/vn
k	后接成分	权责/n 明确/a 的/u 逐级/d 授权/v 制/k
l	习用语	是/v 建立/v 社会主义/n 市场经济/n 体制/n 的/u 重要/a 组成部分/l 。/w
m	数词	科学技术/n 是/v 第一/m 生产力/n
n	名词	希望/v 双方/n 在/p 市政/n 规划/vn
ng	名语素	就此/d 分析/v 时/ng 认为/v
nr	人名	建设部/nt 部长/n 侯/nr 捷/nr
ns	地名	北京/ns 经济/n 运行/vn 态势/n 喜人/a
nt	机构团体	[冶金/n 工业部/n 洛阳/ns 耐火材料/l 研究院/n]nt
nx	字母专名	A T M/nx 交换机/n
nz	其他专名	德士古/nz 公司/n
o	拟声词	汨汨/o 地/u 流/v 出来/v
p	介词	往/p 基层/n 跑/v 。/w
q	量词	不止/v 一/m 次/q 地/u 听到/v ，/w
r	代词	有些/r 部门/n
s	处所词	移居/v 海外/s 。/w
t	时间词	当前/t 经济/n 社会/n 情况/n
tg	时语素	秋/Tg 冬/tg 连/d 旱/a
u	助词	工作/vn 的/u 政策/n
ud	结构助词	有/v 心/n 栽/v 得/ud 梧桐树/n
ug	时态助词	你/r 想/v 过/ug 没有/v
uj	结构助词的	迈向/v 充满/v 希望/n 的/uj 新/a 世纪/n
ul	时态助词了	完成/v 了/ul
Uv	结构助词地	满怀信心/l 地/uv 开创/v 新/a 的/u 业绩/n

续表

代 码	名 称	举 例
uz	时态助词着	眼看/v 着/uz
v	动词	举行/v 老/a 干部/n 迎春/vn 团拜会/n
vd	副动词	强调/vd 指出/v
vg	动语素	做好/v 尊/vg 干/j 爱/v 兵/n 工作/vn
vn	名动词	股份制/n 这种/r 企业/n 组织/vn 形式/n , /w
w	标点符号	生产/v 的/u 5 G/nx 、 /w 8 G/nx 型/k 燃气/n 热水器/n
x	非语素字	生产/v 的/u 5 G/nx 、 /w 8 G/nx 型/k 燃气/n 热水器/n
y	语气词	已经/d 3 0 /m 多/m 年/q 了/y 。 /w
z	状态词	势头/n 依然/z 强劲/a ; /w

以[把][这][篇][报道][编辑][一][下]为例，有  $5 \times 1 \times 1 \times 2 \times 2 \times 2 \times 3 = 120$  种词性标注可能性，如图 4-12 所示，哪种可能性最大？

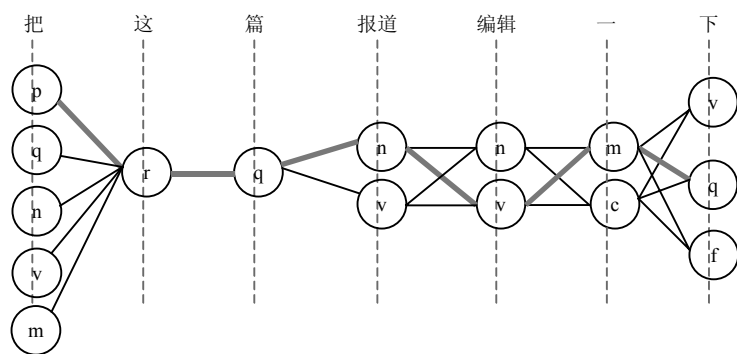


图 4-12 词性标注

解决标注歧义问题最简单的一个方法是从单词所有可能的词性中选出这个词最常用的词性作为这个词的词性，也就是一个概率最大的词性，比如“改革”大部分时候作为一个名词出现，那么可以机械地把这个词总是标注成名词，但是这样标注的准确率会比较低，因为只考虑了频率特征。

考虑词所在的上下文可以提高标注准确率。例如在动词后接名词的概率很大。“推进/改革”中的“推进”是动词，所以后面的“改革”很有可能是名词。这样的特征叫做上下文特征。

隐马尔可夫模型（Hidden Markov Model, HMM）和基于转换的学习方法是两种常用的词性标注方法。这两种方法都整合了频率和上下文两方面的特征来取得好的标注结果。具体来说，隐马尔可夫模型同时考虑到了词的生成概率和词性之间的转移概率。





### 4.10.1 隐马尔可夫模型

词性标注的任务是：给定词序列  $W=w_1, w_2, \dots, w_n$ ，寻找词性标注序列  $T=t_1, t_2, \dots, t_n$ ，最大化概率为  $P(t_1, t_2, \dots, t_n | w_1, w_2, \dots, w_n)$ 。

使用贝叶斯公式重新描述这个条件概率：

$$P(t_1, t_2, \dots, t_n) \times P(w_1, w_2, \dots, w_n | t_1, t_2, \dots, t_n) / P(w_1, w_2, \dots, w_n)$$

忽略掉分母  $P(w_1, w_2, \dots, w_n)$ ，同时做独立性假设，使用  $N$  元模型近似计算  $P(t_1, t_2, \dots, t_n)$ 。例如使用二元连接，则有：

$$P(t_1, t_2, \dots, t_n) \approx \prod_{i=1}^n P(t_i | t_{i-1})$$

近似计算  $P(w_1, w_2, \dots, w_n | t_1, t_2, \dots, t_n)$ ：假设一个类别中的词独立于它的邻居，则有：

$$P(w_1, w_2, \dots, w_n | t_1, t_2, \dots, t_n) \approx \prod_{i=1}^n P(w_i | t_i)$$

寻找最有可能的词性标注序列实际的计算公式：

$$P(t_1, t_2, \dots, t_n) \times P(w_1, w_2, \dots, w_n | t_1, t_2, \dots, t_n) \approx \prod_{i=1}^n P(t_i | t_{i-1}) \times P(w_i | t_i)$$

这里把词  $w$  叫做显状态，词性  $t$  叫做隐状态。条件概率  $P(t_i | t_{i-1})$  叫做转移概率，条件概率  $P(w_i | t_i)$  叫做发射概率。

抽象来说，基本的马尔可夫模型中的状态之间有转移概率。隐马尔可夫模型中有隐状态和显状态。隐状态之间有转移概率。一个隐状态对应多个显状态。隐状态生成显状态的概率叫做生成概率或者发射概率。在初始概率、转移概率以及发射概率已知的情况下，可以从观测到的显状态序列计算出可能性最大的隐状态序列，这个算法叫做维特比（Viterbi）算法。对于词性标注的问题来说，显状态是分词出来的结果——单词  $W$ ，隐状态是需要标注的词性  $C$ 。词性之间存在转移概率。词性按照某个发射概率产生具体的词。可以把初始概率、转移概率和发射概率一起叫做语言模型。因为它们可以用来评估一个标注序列的概率。采用隐马尔可夫模型标注词性的总体结构如图 4-13 所示。

下面举例说明隐马尔可夫模型。假设只有词性：代词（r）、动词（v）、名词（n）和方位词（f）。

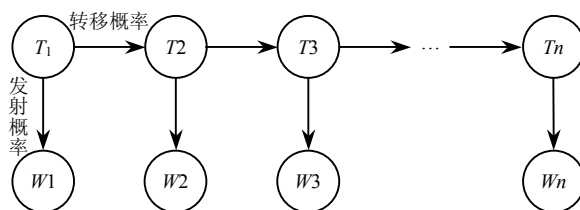


图 4-13 词性标注中的隐马尔可夫模型

有如下一个简化版本的语言模型描述如下：

```
start: go(r,1.0) emit(start,1.0)
f: emit(来,0.1) go(n,0.9) go(end,0.1)
v: emit(来,0.4) emit(会,0.3) go(f,0.1) go(v,0.3) go(n,0.5) go(end,0.1)
n: emit(会,0.1) go(f,0.5) go(v,0.3) go(end,0.2)
r: emit(他,0.3) go(v,0.9) go(n,0.1)
```

这里的 **start** 和 **end** 都是虚拟的状态，**start** 表示开始，**end** 表示结束，**emit** 表示发射概率，**go** 表示转移概率。语言模型中的值可以事前统计出来。中文分词中的语言模型可以从语料库统计出来。

这个语言模型的初始概率向量如表 4-5 所示。

表 4-5 初始概率表

	r	n	f	end
start1	1.0	0	0	0

这个初始概率的意思是，代词是每个句子的开始。

转移概率矩阵如表 4-6 所示。

表 4-6 转移概率表

上个词性 \ 下个词性	start	f	v	n	r	end
start					1	
f				0.9		0.1
v		0.1	0.3	0.5		0.1
n		0.5	0.3			0.2
r			0.9	0.1		

例如第 3 行表示动词后是名词的可能性比较大，仍然是动词的可能性比较小，所以上

个词性是动词，下一个词性是名词的概率是 0.5；而上个词性是动词，下一个词性还是动词的概率是 0.3。

以“他会来”这句话作为例，分词后的输入是：[start][他][会][来][end]。考虑到某些词性更有可能作为一句话的开始，有些词性更有可能作为一句话的结束，这里增加了开始和结束的虚节点[start]和[end]。这句话的转移概率图如图 4-14 所示。

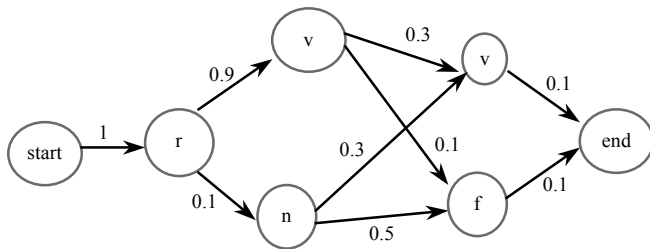


图 4-14 转移概率图

发射概率（混淆矩阵）如表 4-7 所示。

表 4-7 发射概率表

	他	会	来
f			0.1
v		0.3	0.4
n		0.6	
r	0.3		

以发射概率表的第二行为例：如果一个词是动词，那么这个词是“来”的概率比“会”的概率大。例句的发射概率图如图 4-15 所示。

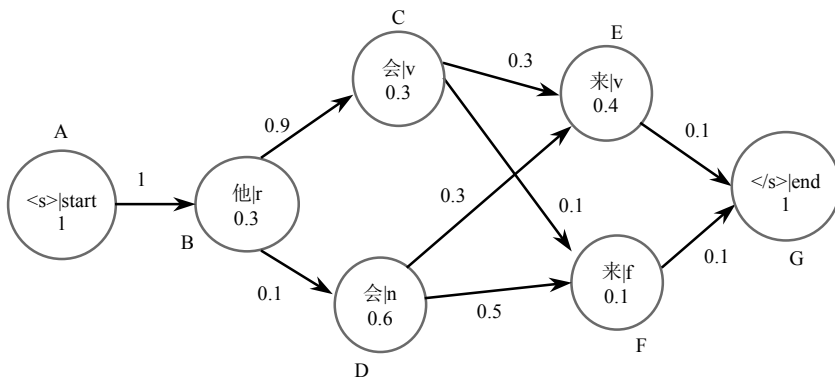


图 4-15 发射概率图

每个隐状态和显状态的每个阶段组合成一个如图 4-16 所示的由节点组成的二维矩阵。

采用动态规划的方法求解最佳标注序列。每个词对应一个求解的阶段，当前节点概率的计算依据是：

- 上一个阶段的节点概率；
- 上一个阶段的节点到当前节点的转移概率；
- 当前节点的发射概率。

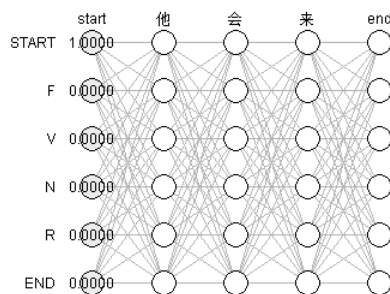


图 4-16 维特比求解格栅

动态规划的思想产生了维特比算法。维特比求解方法由两个过程组成：前向累积概率计算过程和反向回溯过程。前向过程按阶段计算。从图上看就是从前向后按列计算，分别叫做阶段“start”、“他”、“会”、“来”、“end”。

在阶段“start”计算：

- $\text{Best}(A) = 1$

在阶段“他”计算：

- $\text{Best}(B) = \text{Best}(A) \times P(r|\text{start}) \times P(\text{他}|r) = 1 \times 1 \times 0.3 = 0.3$

在阶段“会”计算：

- $\text{Best}(C) = \text{Best}(B) \times P(v|r) \times P(\text{会}|v) = 0.3 \times 0.9 \times 0.3 = 0.081$
- $\text{Best}(D) = \text{Best}(B) \times P(n|r) \times P(\text{会}|n) = 0.3 \times 0.1 \times 0.6 = 0.018$

在阶段“来”计算：

- $\text{Best}(E) = \text{Max} [\text{Best}(C) \times P(v|v), \text{Best}(D) \times P(v|n)] \times P(\text{来}|v) = 0.081 \times 0.3 \times 0.4 = 0.00972$
- $\text{Best}(F) = \text{Max} [\text{Best}(C) \times P(f|v), \text{Best}(D) \times P(f|n)] \times P(\text{来}|f) = 0.081 \times 0.1 \times 0.1 = 0.00081$

在阶段“end”计算：

- $\text{Best}(G) = \text{Max} [\text{Best}(E) \times P(\text{end}|v), \text{Best}(F) \times P(\text{end}|f)] \times P(</s>|\text{end}) = 0.00972 \times 0.1 \times 1 = 0.000972$

执行回溯过程发现最佳隐状态（粗黑线节点），如图 4-17 所示。

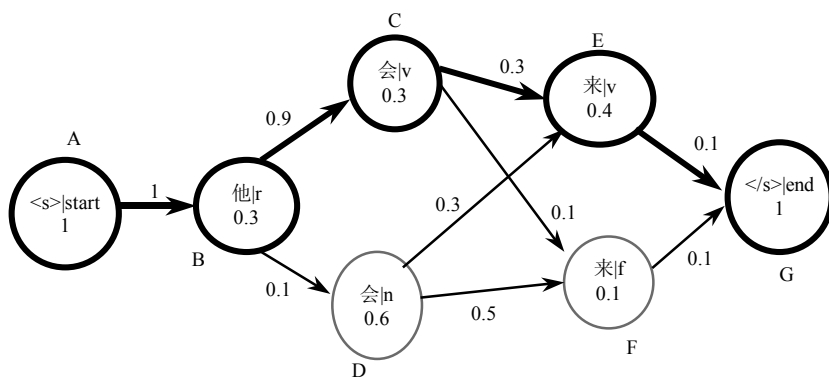


图 4-17 维特比求解过程

G 的最佳前驱节点是 E，E 的最佳前驱节点是 C，C 的最佳前驱节点是 B，B 的最佳前驱节点是 A。所以猜测词性输出：他/r 会/v 来/v。这样消除了歧义，判断出[会]的词性是动词而不是名词，[来]的词性是动词而不是方位词。

然后开始实现维特比算法。维特比算法又叫做在格栅上运行的算法，暗示可以用二维数组存储计算的中间结果，也就是累积概率。但是一个词可能的词性往往只是很少几种，所以一般采用稀疏的方式存储词性。首先定义节点类：

```
public class Node {
    public Node bestPrev; //最佳前驱
    public State tag; //隐状态
    public double prob; //累积概率
}
```

初始化存储累积概率的二维节点：

```
al = new ArrayList[symbols.size()];
for (int i = 0; i < symbols.size(); i++) {
    al[i] = new ArrayList<Node>(states.size());
}
//添加初始节点，即把发射到 start 概率不为零的节点添加到 al[0] 中
for (int j = 0; j < states.size(); j++) {
    double emitProb = states.get(j).emitprob(symbols.get(0));
    if (emitProb > 0) {
        Node tmNode = new Node(states.get(j), emitProb);
        al[0].add(tmNode);
    }
}
//初始化第一阶段以后的节点
```

```

for (int i = 1; i < symbols.size(); i++) {
    for (int j = 0; j < states.size(); j++) {
        double emitProb = states.get(j).emitprob(symbols.get(i));
        if (emitProb > 0) {
            Node tmNode = new Node(states.get(j));
            al[i].add(tmNode);
        }
    }
}

```

维特比求解的前向累积过程主要代码如下所示：

```

//stage 表示显状态序号，跳过 start 状态
for (stage = 1; stage < al.length; stage++) {
    //i1 表示显状态序号 stage 对应的隐状态节点维的序号
    for (i1 = 0; i1 < al[stage].size(); i1++) {
        //i0 表示 i1 上一显状态维的隐状态节点编号
        for (i0 = 0; i0 < al[stage - 1].size(); i0++) {
            sym1 = symbols.get(stage);
            //上一显状态维的隐状态节点
            Node prevNode = al[stage - 1].get(i0);
            //当前显状态维的隐状态节点
            Node nextNode = al[stage].get(i1);
            s0 = prevNode.tag; //上一显状态维的隐状态
            s1 = nextNode.tag; //当前显状态维的隐状态
            //上一显状态维的隐状态节点累积概率
            prob = prevNode.prob;
            //上一显状态维的隐状态到当前显状态维的隐状态的转移概率
            double transprob = s0.transprob(s1);

            prob = prob * transprob;
            emitprob = s1.emitprob(sym1);
            //这一步的累积概率是
            //上一步的累积概率 *
            //上一个状态到这一个状态的转移概率 *
            //当前节点的发射概率
            prob = prob * emitprob;
            //找到当前节点的最大累积概率
            if (nextNode.prob <= prob) {
                //记录当前节点的最大累积概率
                nextNode.prob = prob;
                //记录当前节点的最佳前驱
                nextNode.bestPrev = prevNode;
            }
        }
    }
}

```



```
}  
}
```

维特比求解的反向回溯过程用来寻找最佳路径，主要代码如下所示：

```
public ArrayList<Node> backward() {  
    ArrayList<Node> maxNode = new ArrayList<Node>();  
    Node endNode=al[symbols.size() - 1].get(0);  
    //结束阶段对应的最有可能的隐状态  
  
    //从后往前找最有可能的隐状态  
    for (Node i = endNode; i != null; i = i.bestPrev) {  
        maxNode.add(i); //被选中的隐状态加入到结果路径  
    }  
    return maxNode;  
}
```

统计人民日报语料库中词性间的转移概率矩阵：

```
for (int i = 0; i < wordAndPOS.length - 1; i++) {  
    int j = i + 1;  
    String[] a = wordAndPOS[i].split("/");  
    String[] b = wordAndPOS[j].split("/");  
    int pre = getPOSID(a[1]); //前一个词性 ID  
    int next = getPOSID(b[1]); //后一个词性 ID  
    addWordPOSToMatrix(pre,next); //转移矩阵数组增加一个计数  
}
```

统计某个词的发射概率：

```
public String getFireProbability(CountPOS countPOS) {  
    StringBuilder ret = new StringBuilder();  
    for(Entry<String,Integer> m: posFreqMap.entrySet()) {  
        //某词的这个词性的发射概率是 某词出现这个词性的频率 / 这个词性的总频率  
        double prob =  
            (double)m.getValue() /  
            (double) (countPOS.getFreq(CorpusToDic.getPOSID(m.  
                getKey())));  
        ret.append(m.getKey() + ":" + prob + " ");  
    }  
    return ret.toString();  
}
```

测试一个词的发射概率和相关的转移概率：

```
String testWord = "成果";
System.out.println(testWord+" 的词频率: \n"+this.getWord(testWord));
System.out.println("词性的总频率: \n"+posSumCount);
System.out.println (testWord+" 发射概率: \n"+this.getWord(testWord).
getFireProbability(posSumCount));
System.out.println("转移频率计数取值测试: \n "+this.getTransMatrix
("n", "w"));
printTransMatrix();
```

例如, “成果” 这个词的频率为:

nr:5 b:1 n:287

词性的总频率:

```
a:34578 ad:5893 ag:315 an:2827 b:8734 c:25438 d:47426 dg:125 e:25 f:17279
g:0 h:48 i:4767 j:9309 k:904 l:6111 m:60807 n:229296 ng:4483 nr:35258 ns:27590
nt:3384 nx:415 nz:3715 o:72 p:39907 q:24229 r:32336 s:3850 t:20675 tg:480
u:74751 ud:0 ug:0 uj:0 ul:0 uv:0 uz:0 v:184775 vd:494 vg:1843 vn:42566
w:173056 x:0 y:1900 z:1338
```

“成果” 这个词的发射概率:

nr: 0.00014181178739576834 b: 0.0001144950767117014 n:0.0012516572465285046

“成果” 这个词作为名词的发射概率

= “成果” 作为名词出现的次数/名词的总次数

= 287/229296

= 0.0012516572465285046

为了支持词性标注, 需要在词典中存储词性和对应的次数, 具体格式如下所示:

```
滤波器 n 0
堵击 v 0
稿费 n 7
神机妙算 i 0
开设 vn 0 v 32
```

每行一个词, 然后是这个词可能的词性和语料库中按这个词性出现的次数。存储基本





词性相关信息的类如下所示：

```
public class POSInf {  
    public short pos=0;    //词性，或者词的类别  
    public int freq=0;    //词频，就是一个词在语料库中出现的次数  
                        //词频高就表示这个词是常用词  
    public String seCode = null; //词的语义编码  
}
```

同一个词可以有不同的词性，可以把这些和某个词的词性相关的信息放在同一个链表中。

#### 4.10.2 基于转换的错误学习方法

想象你在画一幅由蓝天、草地、白云组成的油画。先把整块画布涂成蓝色，然后把下面的三分之一用绿色覆盖，表示草地，最后把上面的部分蓝色用白色覆盖，表示白云。这样通过从大到小的修正，越来越接近最终的精细化结果。

基于转换的学习方法（Transformation Based Learning, TBL）先把每个词标注上最可能的词性，然后通过转换规则修正错误的标注，提高标注精度。

一个转换规则的例子：如果一个词左边第一个词的词性是量词（q），左边第二个词的词性是数词（m），则将这个词的词性从动词（v）改为名词（n）。

他/r 做/v 了/u 一/m 个/q 报告/v

转换成：

他/r 做/v 了/u 一/m 个/q 报告/n

另一个转换规则的例子：如果一个词左边第一个词的词性是介词（p），则将这个词的词性从动词（v）改为名词（n）。

从形式上来看，转换规则由激活环境和改写规则两部分组成。例如，对于刚才的例子：

- 改写规则——将一个词的词性从动词（v）改为名词（n）；
- 激活环境——该词左边第一个紧邻词的词性是量词（q），第二个词的词性是数词（m）。

可以从训练语料库中学习出转换规则。学习转换规则序列的过程说明如下。

- ① 初始状态标注：用从训练语料库中统计的最有可能的词性标注语料库中的每个词。
- ② 考查每个可能的转换规则：选择能最多消除语料库标注错误数的规则，把这个规则加到规则序列最后。
- ③ 用选择出来的这个规则重新标注语料库。
- ④ 返回到②，直到标注错误没有明显地减少为止。

这样得到一个转换规则集序列，以及每个词最有可能的词性标注。

标注新数据分两步：首先用最有可能的词性标注每个词，然后依次应用每个可能的转换规则到新数据。

使用基于转换的学习方法标注词性的实现代码如下所示：

```
public List<String> tag(List<String> words) {
    List<String> ret = new ArrayList<String>(words.size());
    for (int i = 0, size = words.size(); i < size; i++) {
        String[] ss = (String[]) lexicon.get(words.get(i));
        if (ss == null)
            ss = lexicon.get(words.get(i).toLowerCase());
        if (ss == null && words.get(i).length() == 1)
            ret.add(words.get(i) + "^");
        if (ss == null)
            ret.add("NN");
        else
            ret.add(ss[0]); //先给每个词标注上最有可能的词性
    }
    //然后依次应用每个可能的转换规则
    for (int i = 0; i < words.size(); i++) {
        String word = ret.get(i);
        // rule 1: DT, {VBD | VBP} --> DT, NN
        if (i > 0 && ret.get(i - 1).equals("DT")) {
            if (word.equals("VBD")
                || word.equals("VBP")
                || word.equals("VB")) {
                ret.set(i, "NN");
            }
        }
    }
}
```



```
    }  
    return ret;  
}
```



## 4.11 平滑算法

语料是有限的，不可能覆盖所有的词汇。比如说  $N$  元模型，当  $N$  较大的时候，由于样本数量有限，导致很多的先验概率值都是 0，这就是零概率问题。当  $N$  值是 1 的时候，也存在零概率问题，例如一些词在词表中，但是却没有出现在语料库中。这说明语料库太小了，没能包括一些本来可能出现的词的句子。

做过物理实验的都知道，我们一般测量了几个点后，就可以画出一条大致的曲线，这叫做回归分析。利用这条曲线，可以修正测量的一些误差，并且还可以估计一些没有测量过的值。平滑算法用观测到的事件来估计未观察到事件的概率。例如从那些比较高的概率值中匀一些给那些低的或者是 0 概率的事件。为了更合理地分配概率，可以根据整个直方图分布曲线去猜那些概率为 0 的事件实际概率值应该是多少。

由于训练模型的语料库规模有限且类型不同，许多合理的搭配关系在语料库中不一定出现，因此会造成模型出现数据稀疏现象。数据稀疏在统计自然语言处理中的一个表现就是零概率问题。有各种平滑算法来解决零概率的问题。例如，我们对自己能做到的事情比较了解，而不太了解别人是否能做到一些事情，这样导致高估自己而低估别人。所以需要开发一个模型减少已经看到的事件的概率，而允许没有看到的事件发生。

平滑有黑盒方法和白盒方法两种。黑盒平滑方法把一个项目作为不可分割的整体；而白盒平滑方法把一个项目作为可分拆的，可用于  $N$  元模型。

加法平滑算法是最简单的一种平滑。加法平滑的原理是给每个项目增加  $\lambda$  ( $1 \geq \lambda \geq 0$ )，然后再除以总数作为项目新的概率。因为数学家拉普拉斯 (Laplace) 首先提出用加 1 的方法估计没有出现过的现象的概率，所以加法平滑也叫做拉普拉斯平滑。

下面是加法平滑算法的一个实现：

```
//根据原始的计数器生成平滑后的分布  
public static <E> Distribution<E> laplaceSmoothedDistribution(  
    GenericCounter<E> counter,  
    int numberOfKeys,
```

```

        double lambda) {
    Distribution<E> norm = new Distribution<E>(); //生成一个新的分布
    norm.counter = new Counter<E>();
    double total = counter.totalDoubleCount(); //原始的出现次数
    double newTotal = total + (lambda * (double) numberOfKeys); //新的出现次数
    //有多大可能性出现零概率事件
    double reservedMass =
        ((double) numberOfKeys - counter.size()) * lambda /
        newTotal;
    norm.numberOfKeys = numberOfKeys;
    norm.reservedMass = reservedMass;
    for (E key : counter.keySet()) {
        double count = counter.getCount(key);
        //对任何一个词来说, 新的出现次数是原始出现次数加 lambda
        norm.counter.setCount(key, (count + lambda) / newTotal);
    }
    if (verbose) {
        System.err.println("unseenKeys=" + (norm.numberOfKeys - norm.
            counter.size()) + " seenKeys=" + norm.counter.size() + " reserved
            Mass=" + norm.reservedMass);
        System.err.println("0 count prob: " + lambda / newTotal);
        System.err.println("1 count prob: " + (1.0 + lambda) / newTotal);
        System.err.println("2 count prob: " + (2.0 + lambda) / newTotal);
        System.err.println("3 count prob: " + (3.0 + lambda) / newTotal);
    }
    return norm;
}
}

```

需要注意的是 **reservedMass** 是所有零概率词出现概率的总和, 而不是其中某个词出现概率的总和。取得指定 **key** 的概率实现代码如下所示:

```

public double probabilityOf(E key) {
    if (counter.containsKey(key)) {
        return counter.getCount(key);
    } else {
        int remainingKeys = numberOfKeys - counter.size();
        if (remainingKeys <= 0) {
            return 0.0;
        } else {
            //如果有零概率的词,
            //则这个词的概率是 reservedMass 分摊到每个零概率词的概率
            return (reservedMass / remainingKeys);
        }
    }
}
}

```

这种方法中的  $\lambda$  值不好选取，在接下来介绍的另外一种平滑算法 Good-Turing 方法中则不需要  $\lambda$  值。为了说明 Good-Turing 方法，首先定义一些标记：假设词典中共有  $x$  个词。在语料库中出现  $r$  次的词有  $N_r$  个。则语料库中的总词数  $N=0*N_0+1*N_1+\dots+r*N_r$ ，而  $x=N_0+N_1+\dots+N_r$ 。

按在语料库中出现的次数分类计算，词概率的最大似然估计值如图 4-18 所示。使用观察到的类别  $r+1$  的全部概率去估计类别  $r$  的全部概率。计算中的第一步是估计语料库中没有见过的词的总概率  $p_0 = N_1/N$ ，分摊到每个词的概率是  $N_1/(N*N_0)$ 。第二步估计语料库中出现过一次的词的总概率  $p_1 = N_2*2/N$ ，分摊到每个词的概率是  $N_2*2/(N*N_1)$ 。以此类推，当  $r$  值比较大时， $N_r$  可能是 0，这时候不再平滑。

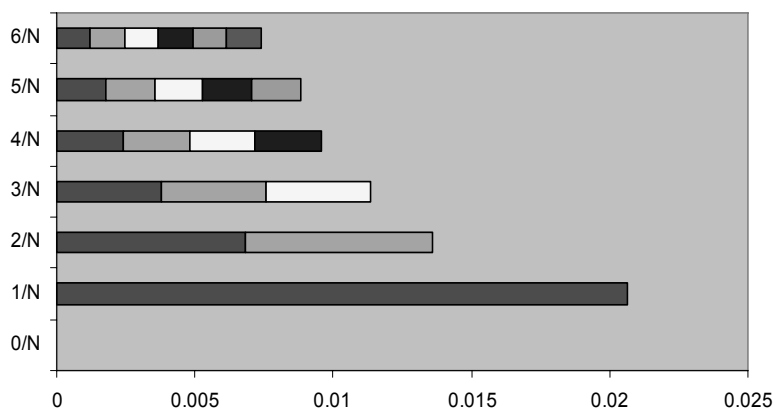


图 4-18 词的频率图

Good-Turing 平滑实现代码如下所示：

```
public static <E> Distribution<E> goodTuringSmoothedCounter(
    GenericCounter<E> counter,
    int numberOfKeys) {
    //收集计数数组，也就是直方图
    int[] countCounts = getCountCounts(counter);

    //如果计数数组不可靠，就不要用 Good-Turing 方法
    //而采用拉普拉斯平滑方法
    for (int i = 1; i <= 10; i++) {
        if (countCounts[i] < 3) {
            return laplaceSmoothedDistribution(counter, numberOfKeys, 0.5);
        }
    }

    double observedMass = counter.totalDoubleCount();
```

```

double reservedMass = countCounts[1] / observedMass;

//计算和缓存调整后的频率，同时也调整观察到的项目的总数
double[] adjustedFreq = new double[10];
for (int freq = 1; freq < 10; freq++) {
    adjustedFreq[freq] = (double) (freq + 1) * (double) countCounts[freq
+ 1] /
                                (double) countCounts[freq];
    observedMass -= ((double) freq - adjustedFreq[freq]) * countCounts
[freq];
}

double normFactor = (1.0 - reservedMass) / observedMass;

Distribution<E> norm = new Distribution<E>();
norm.counter = new Counter<E>();

//填充新的分布，同时重新归一化
for (E key : counter.keySet()) {
    int origFreq = (int) Math.round(counter.getCount(key));
    if (origFreq < 10) {
        norm.counter.setCount(key, adjustedFreq[origFreq] * normFactor);
    } else {
        norm.counter.setCount(key, (double) origFreq * normFactor);
    }
}

norm.numberOfKeys = numberOfKeys;
norm.reservedMass = reservedMass;
return norm;
}

```

对条件概率的 N 元估计平滑：

$$P_{GT}(w_i | w_1, L, w_{i-1}) = \frac{c^*(w_i, w_{i-1})}{c^*(w_1, L, w_{i-1})}$$

这里的  $c^*$  来源于 GT 估计。

例如，估计三元条件概率：

$$P_{GT}(w_3 | w_1, w_2) = \frac{c^*(w_1, w_2, w_3)}{c^*(w_1, w_2)}$$



对于一个没见过的三元联合概率是：

$$P_{GT}(w_1, w_2, w_3) = \frac{C_0^*}{N} = \frac{N_1}{N_o * N}$$

对于一元模型和二元模型也是如此。



## 4.12 本章小结

本章介绍了分词中的查找词典算法。查词典最早用首字母散列或者散列表实现，然后采用 Trie 树的方法开始流行，还有采用数组形式的双数组，后来又发展出和 AC 算法结合的 Trie 树。

因为正向最长匹配最容易实现，所以很多开源的中文分词采用此方法实现。最大概率中文分词是以词为单位的分词，之后出现了按字标注（Character-based Tagging）的 CRF 分词方法。

使用机器学习的方法来标注词性，可以分为两个过程：从训练语料库学习的过程和给一个词序列标注词性的过程。隐马尔可夫模型学习出来的结果是一个语言模型，而基于转换的方法学习出来的结果是一个转换规则集序列，以及每个词最有可能的词性标注。



## 第 5 章

# 让搜索引擎理解自然语言

当用户输入更长的问题时，要能够给出更好的答案。在我们从电影或电视上看到的未来中，搜索引擎已经进化到类似人类助手一样能回答针对任何事物的复杂问题。尽管互联网搜索引擎能够导航非常巨大的知识范围，但是要达到智能助手的能力还很远。一个很明显的不同就是互联网搜索的查询被划分成少数几个关键词而不是以自然语言表述的实际问题。人们可以用基于社区的回答系统通过句子甚至是段落来描述他们的信息需求，因为他们知道如果问题描述得更好，其他人会通读上下文获得更为明确的理解而给出更好的回答。相比而言，一个互联网搜索引擎对于一段很长的查询只能给出很差的返回结果或不返回任何有价值的东西。人们只能将他们的问题转化成一个或几个贴切的关键词去尝试相对贴切的回复。信息检索研究的长期目标是开发检索模型来为更长、更专门的查询提供精确的结果，因此需要更好地理解文本信息。自然语言处理（Natural Language Process）严格来说包括自然语言理解和自然语言生成两部分。这里我们只涉及自然语言理解部分。

如何预测用户查询意图？对于长的查询，可以根据特征词来预测。例如：“感冒了可以吃海鲜吗”中的“吗”字是一个很强的疑问特征词，而“Nokia N97 港行”中“Nokia”、“N97”、“港行”都是很好的产品购买特征词，“松下 328 传真机参数”中的“参数”则是一个很好的产品详细型查询的特征词。





## 5.1 停用词表

在基于词的检索系统中，停用词是指出现频率太高、没有太大检索意义的词，如中文中的“的、了、吧”等，还有英文中的“of、the”等。

可以把这些词放到 stopwords.txt 文本文件中，每行一个词，例如：

的  
了  
吧

创建一个 StopSet 类负责查找一个词是否为停用词。

```
public class StopSet extends HashSet<String>{
    private static final long serialVersionUID = -5739693689170303932L;
    private static StopSet stopSet = new StopSet();

    /**
     *
     * @return the singleton of dictionary
     */
    public static StopSet getInstance(){
        return stopSet;
    }

    //取得词典文件所在的路径
    public static String getDir() {
        String dir = System.getProperty("dic.dir");
        if (dir == null)
            dir = "/dic/";
        else if( !dir.endsWith("/"))
            dir += "/";
        return dir;
    }

    //加载停用词词典
    private StopSet() {
        super(1000);
        String line;
        try{
            String dic = "/stopword.txt";
```

```

InputStream file = null;
if (System.getProperty("dic.dir") == null)
    file = getClass().getResourceAsStream(getDir()+dic);
else
    file = new FileInputStream(new File(getDir()+dic));

BufferedReader in=
    new BufferedReader(new InputStreamReader(file,"GBK"));

while( true )    {
    line = in.readLine();
    if (line == null )
        break;

    if (!"".equals(line))    {
        this.add(line);
    }
}
in.close();
}catch (Exception e)
{
    e.printStackTrace(System.err);
}
}
}

```

然后使用这个停用词表类：

```

Set stopSet=StopSet.getInstance();
if( stopSet.contains(word) ){
    //如果 word 在停用词表中
}

```



## 5.2 句法分析树

句法分析树一般用在机器翻译中，但是搜索引擎也可以借助句法分析树更准确地理解文本，从而更准确地返回搜索结果。比如有用户输入“肩宽的人适合穿什么衣服”，如果返回结果中包括“肩宽的人穿什么衣服好？”或者“肩膀宽的女孩子穿什么衣服好看？”可能是用户想要的结果。

OpenNLP (<http://incubator.apache.org/opennlp/>) 包含一个句法分析树的实现。输入句子

“Boeing is located in Seattle.” 并通过该程序分析后，返回的句法树如图 5-1 所示。

“咬/死/了/猎人/的/狗”这个经过中文分词切分后的句子有两个不同的理解。句法分析能够确定该句子的意义，也就是说句法分析树能消除歧义。

分析树的节点定义如下所示：

```
/** 保存解析构件的数据结构 */
public class Parse {
    /**这个解析基于的文本字符串，同一个句子的
    所有解析共享这个对象*/
    private String text;
    /** 这个构件在文本中代表的字符的偏移量 */
    private Span span;
    /**这个解析的句法类型*/
    private POSType type;
    /** 这个解析的孩子 */
    private Parse[] children;
}
```

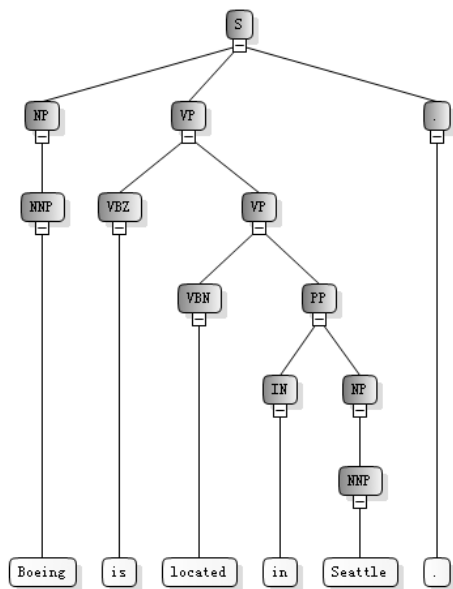


图 5-1 非词汇化的句法树

在句法树的每个节点中还可以增加中心词（head）。中心词就是被修饰部分的词，比如“女教师”，中心词就是“教师”，而“女”就是定语了。为了更好地表示句子中词汇之间的关系，除了在短语结构的句法树中引进中心语，还可以使用表示依存关系的依存树。例如，“这是一本书。”的依存文法树如图 5-2 所示。

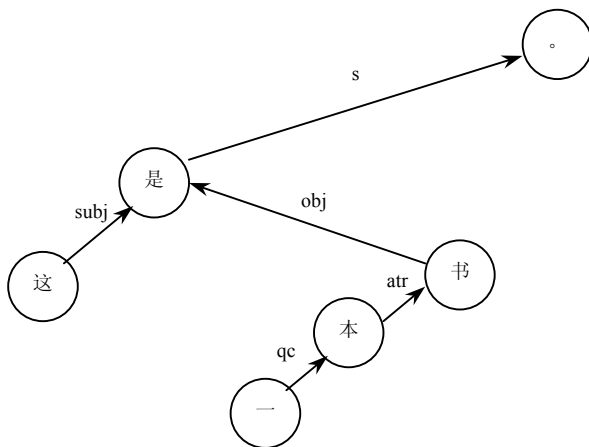


图 5-2 “这是一本书。”的依存文法树

对中文来说，一般先分词然后形成树形结构。

句法分析树可以用自顶向下的方法或者自底向上的方法。`chart parser` 是自顶向下的分析器，`Earley parser` 是 `chart parser` 的一种。

`OpenNLP` 采用了移进-归约（`shift-reduce parser`）的方法实现分析器。移进-归约分析器是一种自底向上的分析器。其基本数据结构是堆栈，检查输入词并且决定是把它移进堆栈还是规约堆栈顶部的元素，把产生式右边的符号用产生式左边的符号替换掉。

移进-归约算法的四种操作说明如下。

- 移进（**Shift**）：从句子左端将一个终结符移到栈顶。
- 归约（**Reduce**）：根据规则，将栈顶的若干个符号替换成一个符号。
- 接受（**Accept**）：句子中所有词语都已移进栈中，且栈中只剩下一个符号 `S`，分析成功，结束。
- 拒绝（**Error**）：句子中所有词语都已移进栈中，栈中并非只有一个符号 `S`，也无法进行任何归约操作，分析失败，结束。

例如表 5-1 所示的产生式列表：

表 5-1 产生式列表

编 号	产 生 式	编 号	产 生 式
1	<code>r -&gt; 我</code>	5	<code>np -&gt; n</code>
2	<code>v -&gt; 是</code>	6	<code>vp -&gt; v np</code>
3	<code>n -&gt; 县长</code>	7	<code>s -&gt; np vp</code>
4	<code>np -&gt; r</code>		

其中 1~3 条可以叫做词法规则（`Lexical rule`），5~7 条叫做内部规则（`Internal rule`）。

使用移进-归约的方法分析词序列“我 是 县长”的过程如表 5-2 所示。

表 5-2 分析词序列“我 是 县长”的过程

栈	输 入	操 作	规 则
	我 是 县长	移进	
我	是 县长	规约	(1) <code>r -&gt; 我</code>
<code>r (1)</code>	是 县长	规约	(4) <code>np -&gt; r</code>
<code>np (4)</code>	是 县长	移进	

续表

栈	输 入	操 作	规 则
np (4) 是	县长	规约	(2) v -> 是
np (4) v (2)	县长	移进	
np (4) v (2) 县长		规约	(3) n -> 县长
np (4) v (2) n (3)		规约	(5) np -> n
np (4) v (2) np (5)		规约	(6) vp -> v np
np (4) vp (6)		规约	(7) s -> np vp
s (7)		接受	

如果在每一步规约的过程中记录父亲指向孩子的引用，则可以生成一个完整的句法树，如图 5-3 所示。

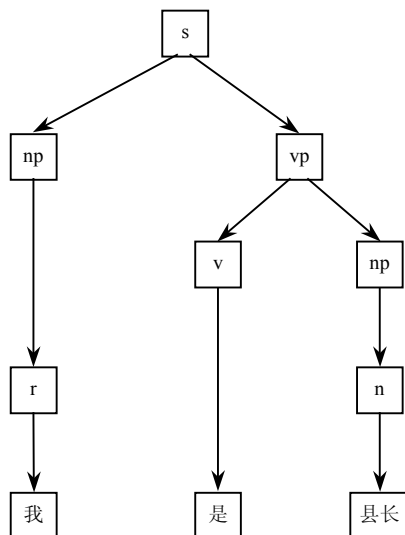


图 5-3 非词汇化的句法树

词汇化规则如表 5-3 所示。

表 5-3 词汇化规则列表

编 号	产 生 式
4	np (我, r) -> r (我, r)
5	np (县长, n) -> n (县长, n)
6	vp (是, v) -> v (是, v) np (县长, n)
7	s (是, v) -> np (我, r) vp (是, v)

根据词汇化规则可以生成如图 5-4 所示的词汇化句法树。

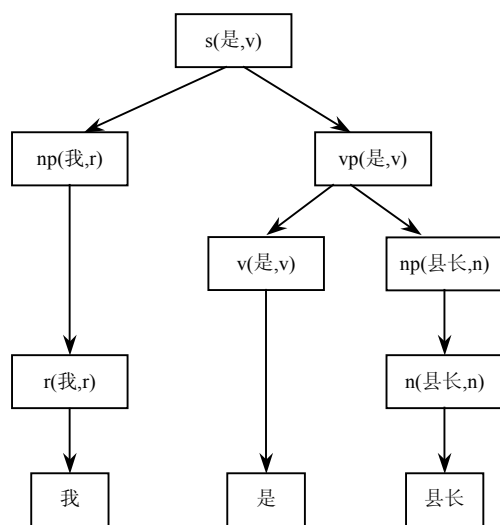


图 5-4 词汇化的句法树

产生式定义如下所示：

```

public class Production {
    protected TokenType lhs; //产生式左边的非终结符
    protected ArrayList<TokenType> rhs; //产生式右边的符号
}

```

基于统计的句法分析训练集是标注了结构的语料库。经过结构标注的语料库叫做树库，如宾夕法尼亚大学树库 (<http://www.cis.upenn.edu/~chinese/>)。

Stanford Parser (<http://nlp.stanford.edu/software/lex-parser.shtml>) 实现了一个基于要素模型的句法分析器，其主要思想就是把一个词汇化的分析器分解成多个要素（factor）句法分析器。Stanford Parser 将一个词汇化的模型分解成一个概率上下文无关文法（PCFG）和一个依存模型。



### 5.3 相似度计算

相似度计算的任务是根据两段输入文本的相似度返回从 0 到 1 之间的相似度值：完全不相似，则返回 0；完全相同，则返回 1。衡量两段文字距离的常用方法有：海明距离（Hamming distance）、编辑距离、欧氏距离、文档向量的夹角余弦距离、最长公共子串。

文档向量的夹角余弦相似度量方法将两篇文档看作是词的向量，如果  $x$ 、 $y$  为两篇文档的向量，则：

$$\text{Cos}(x, y) = \frac{x \cdot y}{\|x\| \|y\|}$$

如果余弦相似度为 1，则  $x$  和  $y$  之间夹角为  $0^\circ$ ，并且除大小（长度）之外， $x$  和  $y$  是相同的；如果余弦相似度为 0，则  $x$  和  $y$  之间夹角为  $90^\circ$ ，并且它们不包含任何相同的词。

衡量文档相似度的另外一种常用方法是最长公共子串法。下面我们举例说明两个字符串  $s1$  和  $s2$  的最长公共子串（Longest Common Subsequence, LCS）。

假设  $s1 = \{a, b, c, b, d, a, b\}$ ， $s2 = \{b, d, c, a, b, a\}$ ，则从前往后找， $s1$  和  $s2$  的最长公共子串是  $\text{LCS}(s1, s2) = \{b, c, b, a\}$ ，如图 5-5 所示。

$s1$  = “高新技术开发区北环海路 128 号” ss

$s2$  = “高技区北环海路 128 号”

则  $s1$  和  $s2$  的最长公共子串为：

$\text{LCS}(s1, s2)$  = “高技区北环海路 128 号”。

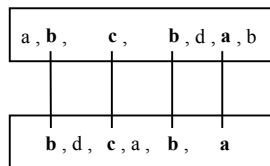


图 5-5 最长公共子串

使用动态规划的思想计算 LCS 的方法说明如下：引进一个二维数组  $\text{num}[][]$ ，用  $\text{num}[i][j]$  记录  $s1$  前  $i$  个长度的子串与  $s2$  前  $j$  个长度的子串的 LCS 长度。

自底向上进行递推计算，那么在计算  $\text{num}[i][j]$  之前， $\text{num}[i-1][j-1]$ 、 $\text{num}[i-1][j]$  与  $\text{num}[i][j-1]$  均已计算出来。此时再根据  $s1[i-1]$  和  $s2[j-1]$  是否相等计算出  $\text{num}[i][j]$ 。

计算两个序列的最长公共子串的实现代码如下所示：

```
public static String longestCommonSubsequence(String s1, String s2){
    int[][] num = new int[s1.length()+1][s2.length()+1]; //初始化为 0 的二维数组

    //实际算法
    for (int i = 1; i <= s1.length(); i++)
        for (int j = 1; j <= s2.length(); j++)
            if (s1.charAt(i - 1)==s2.charAt(j - 1))
                num[i][j] = 1 + num[i-1][j-1];
```

```

        else
            num[i][j] = Math.max(num[i-1][j], num[i][j-1]);

System.out.println("最长公共子串的长度是: " + num[s1.length()][s2.length()]);

int s1position = s1.length(), s2position = s2.length();
StringBuilder result = new StringBuilder();

while (s1position != 0 && s2position != 0) {
    if (s1.charAt(s1position - 1) == s2.charAt(s2position - 1)) {
        result.append(s1.charAt(s1position - 1));
        s1position--;
        s2position--;
    }
    else if (num[s1position][s2position - 1] >= num[s1position - 1][s2position]) {
        s2position--;
    } else {
        s1position--;
    }
}
result.reverse();
return result.toString();
}

```

为了返回 0 到 1 之间的一个相似度值，根据 LCS 计算的打分公式如下：

$$\text{Sim}(s1, s2) = \text{LCS-Length}(s1, s2) / \min(\text{Length}(s1), \text{Length}(s2))$$

最长公共子串（LCS）与夹角余弦相比，最长公共子串体现了词的顺序，而夹角余弦没有。显然，词的顺序在网页文档的相似性比较中是一种重要的信息，一个由若干词语按顺序组成的句子和若干没有顺序的词语组成的集合有着完全不同的意义。完全有可能，两篇文档根本不同，但是夹角余弦值却很接近 1，特别是当文档数规模很大的时候。

比如说“书香门第 4 号门”和“书香门第 4 号”相似度高，但是“书香门第 4 号门”和“书香门第 5 号门”相似度低。所以单从字面上比较无法更准确地反映其相似性。如果只差一个字，还要看有差别的这个字是什么类型的。所以应用带权重的最长公共子串方法，首先将输入字符串切分成词，然后对切分出来的词数组应用带权重的最长公共子序列的相似度打分算法。让比较的元素 E 增加权重属性 Weight。

```

public static double addSim(String n1, String n2) throws Exception {
    // 首先执行同义词替换，例如把“地方税务局”替换成“地税局”
}

```





```
String s1 = SynonymReplace.replace(n1);
String s2 = SynonymReplace.replace(n2);

double d = getSim(s1, s2);
if (d > 0.7) {
    ArrayList<PoiToken> ret1 = PoiTagger.basicTag(s1);
    ArrayList<PoiTokenWeight> poiW1 = new ArrayList<PoiTokenWeight>();

    for (int i = 0; i < ret1.size(); i++) {
        PoiToken poi = ret1.get(i);
        PoiTokenWeight poiTokenWeight = new PoiTokenWeight(poi);
        poiW1.add(poiTokenWeight);
    }
    ArrayList<PoiToken> ret2 = PoiTagger.basicTag(s2);
    ArrayList<PoiTokenWeight> poiW2 = new ArrayList<PoiTokenWeight>();

    for (int i = 0; i < ret2.size(); i++) {
        PoiToken poi = ret2.get(i);
        PoiTokenWeight poiTokenWeight = new PoiTokenWeight(poi);
        poiW2.add(poiTokenWeight);
    }
    return getSim(poiW1, poiW2);
}
return d;
}
//应用带权重的 LCS
double getSim(ArrayList<PoiToken>s1, ArrayList<PoiToken>s2){
    ...
    if (s1.get(i-1).equals(s2.get(j-1))) {
        num[i][j] = s1.get(i-1).Weight + num[i-1][j-1]; //增加权重
    } else{
        num[i][j] = Math.max(num[i-1][j], num[i][j-1]);
    }
    ...
}
```

上述的相似度计算方法中没有考虑词语之间的语义相关度。例如，“国道”和“高速公路”在字面上不相似，但是两个词在意义上有相关性。可以使用分类体系的语义词典提取词语语义相关度。基于语义词典度量方法的计算公式，以下因素是最经常使用的：

- 最短路径长度，即两个概念节点 A 和 B 之间所隔最少的边数量。
- 局部网络密度，即从同一个父节点引出的子节点数量。显然，层次网络中各个部分的密度是不相同的。例如，WordNet 中的 plant/flora 部分是非常密集的，一个父节

点包含了数百个子节点。对于一个特定节点（和它的子节点）而言，全部的语义块是一个确定的数量，所以局部密度越大，节点（即父子节点或兄弟节点）之间的距离越近。

- 节点在层次中的深度。在层次树中，自顶向下，概念的分类是由大到小，大类间的相似度肯定要小于小类间的相似度。所以当概念由抽象逐渐变得具体，连接它们的边对语义距离计算的影响应该逐渐减小。
- 连接的类型，即概念节点之间的关系类型。在许多语义网络中，上下位关系是一种最常见的关系，所以许多基于边的方法也仅仅考虑 IS-A 连接。事实上，如果可以得到其他类型的信息，如部分关系和整体关系，那么其他的关系类型对于边权重计算的影响也同样应该考虑。
- 概念节点的信息含量。它的基本思想是用概念间的共享信息作为度量相似性的依据，方法是从语义网中获得概念间的共享信息，从语料库的统计数据中获得共享信息的信息量，综合两者计算概念间的相似性。这种方法基于一个假设：概念在语料库中出现的频率越高，则越抽象，信息量越小。
- 概念的释义。在基于词典的模型中——不论是基于传统词典，还是基于语义词典——词典被视为一个闭合的自然语言解释系统，每一个单词都被词典中其他的单词所解释。两个单词的释义词汇集重叠程度越高，则表明这两个单词越相似。

将上述六个因素进一步合并，则可归为三大因素：结构特点、信息量和概念释义。



## 5.4 文档排重

不同的网站间转载内容的情况很常见。即使在同一个网站，有时候不同的 URL 地址可能对应同一个页面，或者存在同样的内容以多种方式显示出来，所以网页需要按内容做文档排重。

判断文档的内容重复有很多种方法，语义指纹的方法比较高效。语义指纹直接提取一个文档的二进制数组表示的语义，通过比较相等来判断网页是否重复。语义指纹是一个很大的数组，全部存放在内存会导致内存溢出，普通的数据库效率太低，所以这里采用内存数据库 BerkeleyDB。可以通过 BerkeleyDB 判断该语义指纹是否已经存在。另外一种方法是通过第 2 章介绍过的布隆过滤器来判断语义指纹是否重复。

按词作维度的文档向量维度很高，可以把 SimHash 看成是一种维度削减技术。SimHash



除了可以用在文档排重上，还可以用在任何需要计算文档之间的距离应用上，例如文本分类或聚类。

### 5.4.1 语义指纹

提取网页的语义指纹的方法是：对于每张网页，从净化后的网页中，选取最有代表性的一组关键词，并使用该关键词组生成一个语义指纹。通过比较两个网页的指纹是否相同来判断两个网页是否相似。

网络一度出现过很多关于“罗玉凤征婚”的新闻报道。其中的两篇新闻对比如表 5-4 所示。

表 5-4 两篇新闻对比

文档 ID	文档 1	文档 2
标题	非北大清华硕士不嫁的“最牛征婚女”	1 米 4 专科女征婚 求 1 米 8 硕士男 应征者如云
内容	...24 岁的罗玉凤，在上海街头发放了 1300 份征婚传单。传单上写了近乎苛刻的条件，要求男方北大或清华硕士，身高 1 米 76 至 1 米 83 之间，东部沿海户籍。而罗玉凤本人，只有 1 米 46，中文大专学历，重庆綦江人。...此事经网络曝光后，引起了很多人的兴趣。“每天都有打电话、发短信求证，或者是应征。”罗玉凤说，她觉得满意的却寥寥无几，“到目前为止只有 2 个，都还不是特别满意”。...	...24 岁的罗玉凤，在上海街头发放了 1300 份征婚传单。传单上写了近乎苛刻的条件，要求男方北大或清华硕士，身高 1 米 76 至 1 米 83 之间，东部沿海户籍。而罗玉凤本人，只有 1 米 46，中文大专学历，重庆綦江人。...此事经网络曝光后，引起了很多人的兴趣。“每天都有打电话、发短信求证，或者是应征。”罗玉凤说，她觉得满意的却寥寥无几，“到目前为止只有 2 个，都还不是特别满意”。...

对于这两篇内容相同的新闻，有可能提取出同样的关键词：“罗玉凤”、“征婚”、“北大”、“清华”、“硕士”，这就表示这两篇文档的语义指纹也相同。

为了提高语义指纹的准确性，需要考虑到同义词，例如：“北京华联”和“华联商厦”可以看成相同意义的词，可以做同义词替换。把“开业之初，比这还要多的质疑的声音环绕在北京华联决策者的周围”替换为“开业之初，比这还要多的质疑的声音环绕在华联商厦决策者的周围”。

设计同义词词典的格式是：每行一个义项，前面是基本词，后面是一个或多个被替换的同义词，例如：

华联商厦 北京华联 华联超市

这样会把“北京华联”和“华联超市”替换成“华联商厦”。对指定文本，要从前往后

查找同义词词库中每个要替换的词，然后实施替换。同义词替换的实现代码分为两步。首先是查找 Trie 树结构的词典过程：

```
public void checkPrefix(String sentence,int offset,PrefixRet ret) {
    if (sentence == null || root == null || "".equals(sentence)) {
        ret.value = Prefix.MisMatch;
        ret.data = null;
        ret.next = offset;
        return ;
    }
    ret.value = Prefix.MisMatch;//初始返回值设为没匹配上任何要替换的词
    TSTNode currentNode = root;
    int charIndex = offset;
    while (true) {
        if (currentNode == null) {
            return;
        }
        int charComp = sentence.charAt(charIndex) - currentNode.splitchar;

        if (charComp == 0) {
            charIndex++;

            if(currentNode.data != null){
                ret.data = currentNode.data;//候选最长匹配词
                ret.value = Prefix.Match;
                ret.next = charIndex;
            }
            if (charIndex == sentence.length()) {
                return; //已经匹配完
            }
            currentNode = currentNode.eqKID;
        } else if (charComp < 0) {
            currentNode = currentNode.loKID;
        } else {
            currentNode = currentNode.hiKID;
        }
    }
}
```

然后是同义词替换过程：

```
//输入待替换的文本，返回替换后的文本
public static String replace(String content) throws Exception{
    int len = content.length();
    StringBuilder ret = new StringBuilder(len);
```



```
SynonymDic.PrefixRet matchRet = new SynonymDic.PrefixRet(null,null);

for(int i=0;i<len;){
    //检查是否存在从当前位置开始的同义词
    synonymDic.checkPrefix(content,i,matchRet);
    if(matchRet.value == SynonymDic.Prefix.Match)
        //如果匹配上，则替换同义词
    {
        ret.append(matchRet.data);//把替换词输出到结果
        i=matchRet.next;//下一个匹配位置
    }
    else //如果没有匹配上，则从下一个字符开始匹配
    {
        ret.append(content.charAt(i));
        ++i;
    }
}

return ret.toString();
}
```

语义指纹生成算法说明如下。

- ① 将每个网页分词表示成基于词的特征向量，使用 TF\*IDF 作为每个特征项的权值。地名，专有名词等，名词性的词汇往往有更高的语义权重。
- ② 将特征项按照词权值排序。
- ③ 选取前  $n$  个特征项，然后重新按照字符排序。如果不排序，关键词就找不到对应关系。
- ④ 调用 MD5 算法，将每个特征项串转化为一个 128 位的串，作为该网页的指纹。

调用 `fseg.result.FingerPrint` 中的方法：

```
String fingerPrint = getFingerPrint("", "昨日，省城渊明北路一名 17 岁的少年在 6 楼晾毛巾时失足坠楼，摔在楼下的一辆面包车上。面包车受冲击变形时吸收了巨大的反作用力能量，从而“救”了少年一命。目前，伤者尚无生命危险。 据一位目击者介绍，事故发生在下午 2 时 40 分许，当时这名在某美发店工作的少年正站在阳台上晾毛巾，因雨天阳台湿滑而不小心摔下。记者来到抢救伤者的医院了解到，这名少年名叫李嘉诚，今年 17 岁，系丰城市人。李嘉诚受伤后，他表姐已赶到医院陪护。据医生介绍，伤者主要伤在头部，具体伤情还有待进一步检查。");
String md5Value = showBytes(getMD5(fingerPrint));
System.out.println("FingerPrint:"+fingerPrint+" md5:"+md5Value);
```

MD5 可以将字符串转化成几乎无冲突的 hash 值。但是 MD5 速度比较慢，MurmurHash 或者 JenkinsHash 也可以生成冲突很少的 hash 值，在 Lucene 的企业搜索软件 Solr1.4 版本中提供了 JenkinsHash 实现的语义指纹，叫做 Lookup3Signature。调用 MurmurHash 生成 64 位 hash 的代码如下所示：

```
public static long stringHash64(String str, int initial) {
    byte[] bytes = str.getBytes();
    return MurmurHash.hash64(bytes, initial);
}
```

### 5.4.2 SimHash

根据上面这个 MD5 方法的语义指纹无法找出特征近似的文档。例如，对于两个相似的文档，其 MD5 值是完全不同的。关键字的微小差别会导致 MD5 的 hash 值差异巨大，这是 MD5 算法中雪崩效应（avalanche effect）的结果。输入中一位的变化，散列结果中将有一半以上的位改变。

如果两个相似文档的语义指纹只相差几位或更少，这样的语义指纹叫做 SimHash。可以用海明距离来衡量近似的语义指纹。海明距离是针对长度相同的字符串或二进制数组而言的。对于二进制数组  $s$  和  $t$ ， $H(s, t)$  是两个数组对应位有差别的数量。例如：1011101 和 1001001 的海明距离是 2。下面的方法可以按位比较计算两个 64 位的长整型之间的海明距离。

```
public static int hamming(long l1, long l2) {
    int counter = 0;
    for (int c=0; c<64; c++)
        counter += (l1 & (1L << c)) == (l2 & (1L << c)) ? 0 : 1;
    return counter;
}
```

这种按位比较的方法比较慢，可以把两个长整型按位异或（XOR），然后计算结果中 1 的个数，结果就是海明距离。例如计算 A 和 B 的海明距离：

A = 1 1 1 0

B = 0 1 0 0

A XOR B = 1 0 1 0

计算 1 0 1 0 中 1 的个数是 2，实现代码如下所示：

```
public static int hammingXOR(long l1, long l2) {
    long lxor = l1 ^ l2; //按位异或
    return BitUtil.pop(lxor); //计算 1 的个数
}
```

假设可以得到文档的一系列特征，每个特征有不同的重要度。计算文档对应的 SimHash 值的方法是把每个特征的 Hash 值叠加到一起形成一个 SimHash。计算过程如图 5-6 所示。

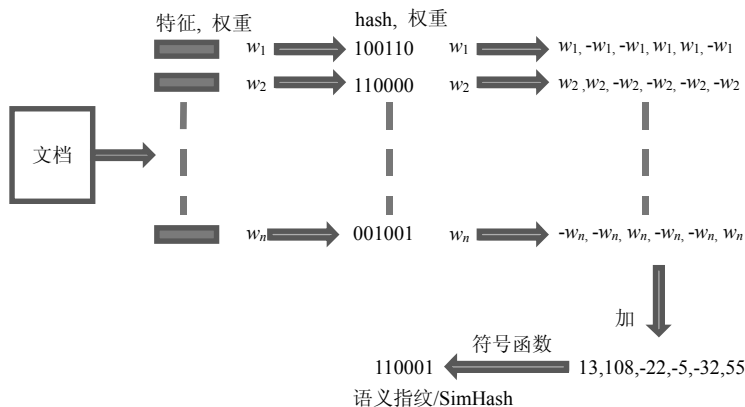


图 5-6 语义指纹计算过程

可以把特征权重看成特征在 SimHash 结果的每一位上的投票权。权重大的特征投票权大，权重小的特征投票权小。所以权重大的特征更有可能影响文档 SimHash 值中的很多位，而权重小的特征影响文档 SimHash 值的位数很少。

假定 SimHash 的长度为 64 位，文档的 SimHash 计算过程说明如下。

① 初始化长度为 64 位的数组，该数组的每个元素都是 0。

② 对于特征列表循环做如下处理：

① 取得每个特征的 64 位的 hash 值。

② 如果这个 hash 值的第  $i$  位是 1，则将数组的第  $i$  个数加上该特征的权重；反之，如果 hash 值的第  $i$  位是 0，则将数组的第  $i$  个数减去该特征的权重。

③ 完成所有特征的处理，数组中的某些数为正，某些数为负。SimHash 值的每一位与数组中的每个数对应，将正数对应的位设为 1，负数对应的位设为 0，就得到 64 位的 0/1 值的位数组，即最终的 SimHash。

输入特征和权重数组，返回 SimHash 的代码如下所示：

```
public static long simHash(String[] features,int[] weights){
    int[] hist = new int[64]; //创建直方图

    for(int i=0;i<features.length;++i) {
        long addressHash = stringHash(features[i]); //生成特征的 hash 码
        int weight = weights[i];
        /* 更新直方图 */
        for (int c=0; c<64; c++)
            hist[c] += (addressHash & (1 << c)) == 0 ? -weight : weight;
    }

    /* 从直方图计算位向量 */
    long simHash=0;
    for (int c=0; c<64; c++) {
        long t= ((hist[c]>=0)?1:0);
        t <<= c;
        simHash |= t ;
    }

    return simHash;
}
```

要生成好的 SimHash 编码，就要让完全不同的特征差别尽量大，而相似的特征差别比较小。如果特征是枚举类型，只有两个可能的取值，例如是 Open 和 Close。Open 返回二进制位全是 1 的散列编码，而 Close 则返回二进制位全是 0 的散列编码。下面的代码将为指定的枚举值生成尽量不一样的散列编码。

```
public static long getSimHash(MatterType matter){
    int b=1; //记录用多少位编码可以表示一个枚举类型的集合
    int x=2;
    while(x<MatterType.values().length) {
        b++;
        x = x<<1;
    }

    long simHash = matter.ordinal();
    int end = 64/b;
    for(int i=0;i<end;++i) {
        simHash = simHash << b; //枚举值按枚举类型总个数向左移位
        simHash += matter.ordinal();
    }
    return simHash;
}
```





中文字符串特征的散列算法如下所示：

```
public static int byte2int(byte b) {
    return (b & 0xff);
}

private static int MAX_CN_CODE = 6768; //最大中文编码
private static int MAX_CODE = 6768+117; //最大编码

//取得中文字符的散列编码
public static int getHashCode(char c) throws UnsupportedOperationException{
    String s = String.valueOf(c);
    int maxVal = 6768;
    byte[] b = s.getBytes("gb2312");
    if(b.length==2) {
        int index = (byte2int(b[0]) - 176) * 94 + (byte2int(b[1]) - 161);
        return index;
    }
    else if(b.length==1) {
        int index = byte2int(b[0]) - 9 + MAX_CN_CODE;
        return index;
    }
    return c;
}

//取得中文字符串的散列编码
public static long getSimHash(String input) throws UnsupportedOperationException{
    if(input==null || "".equals(input)) {
        return -1;
    }
    int b=13; //记录用多少位编码可以表示一个中文字符

    long simHash = getHashCode(input.charAt(0));
    int maxBit = b;
    for(int i=1;i<input.length();++i) {
        simHash *= MAX_CODE; //把汉字串看成是 MAX_CODE 进制的
        simHash += getHashCode(input.charAt(i));
        maxBit += b;
    }

    long originalValue = simHash;

    for(int i=0;i<=(64/maxBit);++i){
        simHash = simHash << maxBit;
        simHash += originalValue;
    }
}
```

```

return simHash;
}

```

SimHash 的计算依据是待比较对象的特征，对于结构化的记录可以按列提取特征，而非结构化的文档特征则不明显。如果是新闻，特征可以用标题或最长的几句话来表示。提取特征前，最好先进行一些简单的预处理，如全角转半角。

基于 SimHash 的文档排重流程如图 5-7 所示。

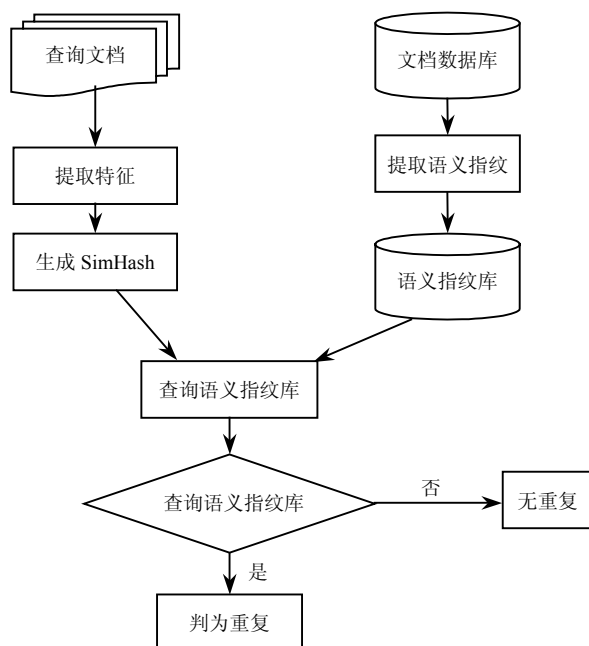


图 5-7 文档排重计算过程

根据文档排重流程设计出对结构化记录使用排重接口的方式：首先根据文档集合生成一个语义指纹的集合（fingerprintSet），然后根据待查重的文档生成一个 SimHash 值来查找近似的文档集合。fingerprintSet.getSimSet(key, k) 返回和 key 相似的数据集合。

```

//返回一条记录的 simHash
long simHashKey = poiSimHash.getHash(poi,address,tel);
//根据一条记录的 simHash 返回相似的数据
HashSet<SimHashData> ret = fingerprintSet.getSimSet(simHashKey, k);

```

把文档转换成 SimHash 后，文档排重就变成了计算海明距离的问题。海明距离计算问题是：给出一个  $f$  位的语义指纹集合  $F$  和一个语义指纹  $fg$ ，找出  $F$  中是否存在与  $fg$  只有  $k$



位差异的语义指纹。

最基本的一种方法是逐次探查法，先把所有和  $fg$  差  $k$  位的指纹找出来，然后用折半查找法查找排好序的指纹集合  $F$ 。需要多少次折半查找呢？首先借助组合数生成器 `CombinationGenerator` 来生成和给定的语义指纹差别在 2 位以内的语义指纹：

```
long fingerPrint = 1L; //语义指纹
int[] indices; //组合数生成的一种组合结果
//生成差 2 位的语义指纹
CombinationGenerator x = new CombinationGenerator(64, 2);
int count = 0; //计数器
while (x.hasMore()) {
    indices = x.getNext(); //取得组合数生成结果
    long simFP = fingerPrint;
    for (int i = 0; i < indices.length; i++) {
        simFP = simFP ^ 1L << indices[i]; //翻转对应的位
    }
    System.out.println(Long.toBinaryString(simFP)); //打印相似语义指纹
    ++count;
}
```

这里运行的结果是 `count=2016`。因为是从 64 位中选有差别的 2 位，所以计算公式是  $C_{64}^2 = 64 \times 63 / 2 = 2016$ 。也就是说，要找出和给定语义指纹差别在 2 位以内的语义指纹需要探查 2016 次。

逐次探查法完整的查找过程如下所示：

```
//输入要查找的语义指纹和 k 值，如果找到相似的语义指纹则返回真，否则返回假
public boolean containSim(long fingerPrint, int k) {
    //首先用二分法直接查找语义指纹
    if (contains(fingerPrint)) {
        return true;
    }

    //然后用逐次探查法查找
    int[] indices;

    for (int ki = 1; ki <= k; ++ki) {
        //找差 1 位直到差 k 位的
        CombinationGenerator x = new CombinationGenerator(64, ki);
        while (x.hasMore()) {
            indices = x.getNext();
            long simFP = fingerPrint;
            for (int i = 0; i < indices.length; i++) {
```

```

        simFP = simFP ^ 1L << indices[i];
    }
    //查找相似语义指纹
    if(contains(simFP)) {
        return true;
    }
}

return false;
}

```

在  $k$  值很小而要找的语义指纹集合  $S$  中的元素不太多的情况下，可以用比逐次探查法更快的方法查找。

如果  $k$  值很小，例如  $k=1$ ，可以给指纹集合  $S$  中每个元素生成出和这个元素差别在 1 位以内的元素。对于长整型的元素，差别在 1 位以内的元素只有 65 种可能。然后再把所有这些新生成的元素排序，最后用折半查找算法查询这个排好序的语义指纹集合  $\text{simSet}$ 。生成法查找近似语义指纹的整体流程如图 5-8 所示。

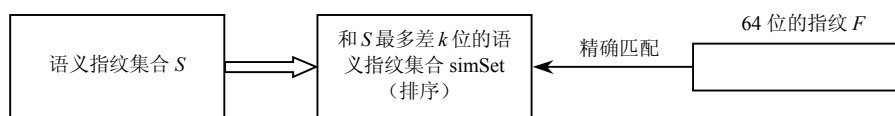


图 5-8 生成法查找近似语义指纹

对给定 SimHash 值  $n$  生成差 1 位的 Hash 值的代码如下所示：

```

for(int j=0;j<64;j++){
    long newSimHash=n^1L<<j; //生成和 n 差 1 位的 Hash 值
}

```

在  $k$  值比较小的情况下，例如  $k$  不大于 3，介绍另外一种快速计算方法。假设我们有一个已经排序的容量为  $2^d$  的  $f$  位指纹集合。看每个指纹的高  $d$  位。该高  $d$  位具有以下性质：因为指纹集合中有很多的位组合存在，所以高  $d$  位中只有少量重复。

SimHash 排重有以下假设。

- 整个表中排列组合的部分很少，不太可能出现如“一批 8 位 SimHash，前 4 位都一样，但后 4 位出现 16 种 0-1 组合”的情况。
- 整个表在前  $d$  位 0-1 分布不会有很多的重复。



这两个假设得到排重的基础：

- 前  $d$  位上的 0-1 分布足以当成一个指针；
- 有能力快速搜索前  $d$  位。

现在找一个接近于  $d$  的数字  $d'$ ，由于整个表是排好序的，所以一趟搜索就能找出高  $d'$  位与目标指纹  $F$  相同的指纹集合  $f'$ 。因为  $d'$  和  $d$  很接近，所以找出的集合  $f'$  也不会很大。

$$|f'| = |S|/2^{d'}$$

最后在集合  $f'$  中查找和  $F$  之间海明距离为  $k$  的指纹也就很快了。总的思想是：先把检索的集合缩小  $2^{d'}$  倍，然后在小集合中逐个检查，看剩下的  $f-d'$  位的海明距离是否满足要求。假设  $k$  值为 3，有  $2^{34}$  个语义指纹待查找。

**第一种策略：** $f$  分为 6 块，分别是 11、11、11、11、10、10 位，最坏的可能是其中 3 块里各出现 1 位海明距离不同，把这 3 块限制到低位，换言之，选 3 块精确匹配的到高位，有  $C_6^3 = 20$  种选法，因此需要复制 20 个 T 表。对每个表高位做精确匹配，需要匹配 31~33 位，(11×3、11×2+10、11+10×2) 那么  $f'$  的个数大概是  $|S|/2^{31} = 2^{[34-31]} = 8$ ，即精确匹配一次，产生大约 8 个需要计算海明距离的 SimHash。

**第二种策略：** $f$  先分为 4 块，各 16 位，选一个精确匹配块到高位的可能有  $C_4^1 = 4$  种选法，再对剩下 3 块 48 位切分，分成 4 块，各 12 位，选一个精确匹配块到高位的可能有  $C_4^1 = 4$  种， $4 \times 4 = 16$ ，一共要复制 16 次 T 表，那么高位就有  $16+12 = 28$  位。每次精确匹配 28 位后，产生大约  $2^{[34-28]} = 64$  个需要计算  $d'$  海明距离的 SimHash。

假设 SimHash 有  $f$  位，现在找一个接近于  $d$  的数字  $d'$ ，由于整个表是排好序的，所以一趟精确匹配就能找出高  $d'$  位与目标指纹  $F$  相同的指纹集合  $f'$  ( $f' = 2^{[d-d']}$ )，因为  $d'$  和  $d$  很接近，所以找出的集合  $f'$  也不会很大。海明距离的比较就在  $f-d'$  位上进行。要确保海明位不同的几位都被限制在  $f-d'$  上就需要考虑  $f$  上不同位的组合可能，即让海明位不会出现在前  $d'$  位上，每种  $f$  位上的组合就需要复制一次 T。

该算法本质就是采用分治法，把问题分解成更小的几个子问题，降低问题需要处理的数据规模。利用空间（原空间的  $t$  倍）和并行计算换时间。分治法查找海明距离在  $k$  以内的语义指纹算法步骤说明如下。

- ① 复制原表  $T$  为  $Tt$  份： $T1, T2, \dots, Tt$ 。

② 每个  $T_i$  都关联一个  $pi$  和一个  $\pi i$ ，其中  $pi$  是一个整数， $\pi i$  是一个置换函数，负责把  $pi$  个位换到高位上。

③ 应用置换函数  $\pi i$  到相应的  $T_i$  表上，然后对  $T_i$  进行排序。

④ 对每一个  $T_i$  和要匹配的指纹  $F$ 、海明距离  $k$  做如下运算：使用  $F$  的高  $pi$  位检索，找出  $T_i$  中高  $pi$  位相同的集合，在检索出的集合中比较剩下的  $f-pi$  位，找出海明距离小于或等于  $k$  的指纹。

⑤ 合并所有  $T_i$  中检索出的结果。

举一个实现的例子，假设有 100 亿左右 ( $2^{34}$ ) 的语义指纹，SimHash 有 64 位，所以  $f=64$ ， $d=34$ ，海明距离  $k$  值是 3。

例如，SimHash 长度是 64 位，按 16 位拆分，复制 4 份，分别是  $T_1$ 、 $T_2$ 、 $T_3$ 、 $T_4$ 。这里，在  $T_{i-1}$  的基础上左移 16 位，精确匹配每个复制表的高 16 位，然后在精确匹配出来的结果中找差 3 位以内的 SimHash。按 16 位拆分的查找方法如图 5-9 所示。

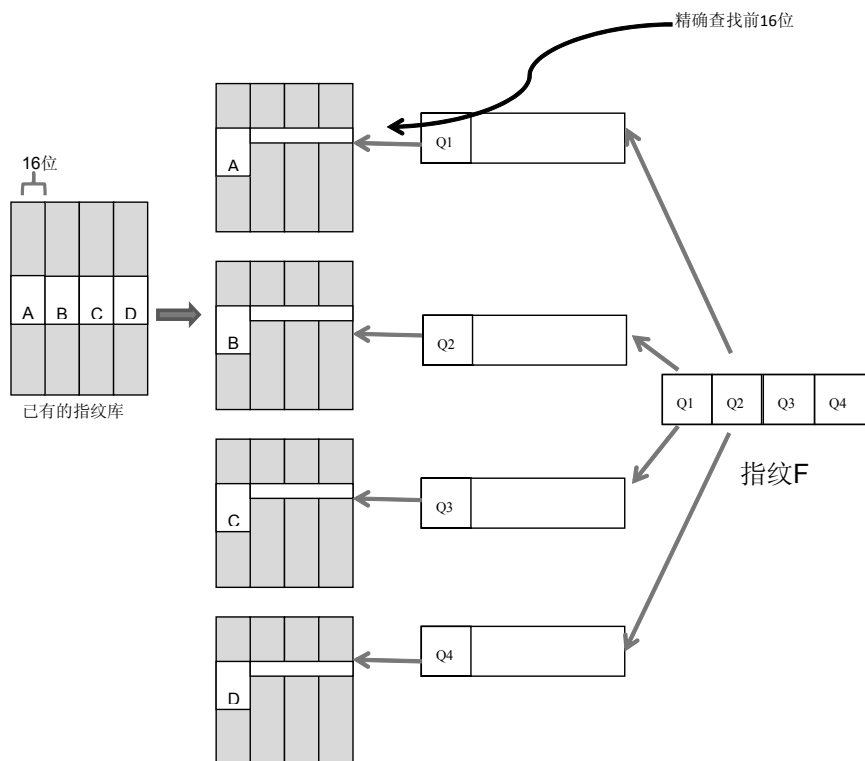


图 5-9 分 4 块查找语义指纹



比较用长整型表示的无符号 64 位语义指纹的代码如下所示：

```
public static boolean isLessThanUnsigned(long n1, long n2) {
    return (n1 < n2) ^ ((n1 < 0) != (n2 < 0));
}

static Comparator<SimHashData> comp = new Comparator<SimHashData>(){
    public int compare(SimHashData o1, SimHashData o2){
        if(o1.q==o2.q) return 0;
        return (isLessThanUnsigned(o1.q,o2.q)) ? 1: -1;
    }
}; //比较无符号 64 位

static Comparator<Long> compHigh = new Comparator<Long>(){
    public int compare(Long o1, Long o2){
        o1 |= 0xFFFFFFFFFFFFL;
        o2 |= 0xFFFFFFFFFFFFL;
        //System.out.println(Long.toBinaryString(o1));
        //System.out.println(Long.toBinaryString(o2));
        //System.out.println((o1 == o2));
        if(o1.equals(o2)) return 0;
        return (isLessThanUnsigned(o1,o2)) ? 1: -1;
    }
}; //比较无符号 64 位中的高 16 位

public void sort(){//对四个表排序
    t2.clear();
    t3.clear();
    t4.clear();
    for(SimHashData simHash:t1){
        long t = Long.rotateLeft(simHash.q, 16);
        t2.add(new SimHashData(t,simHash.no));

        t = Long.rotateLeft(t, 16);
        t3.add(new SimHashData(t,simHash.no));

        t = Long.rotateLeft(t, 16);
        t4.add(new SimHashData(t,simHash.no));
    }

    Collections.sort(t1, comp);
    Collections.sort(t2, comp);
    Collections.sort(t3, comp);
    Collections.sort(t4, comp);
}
```

### 5.4.3 分布式文档排重

在批量版本的海明距离问题中，有一批查询语义指纹，而不是一个查询语义指纹。

假设已有的语义指纹库存储在文件 F 中，批量查询语义指纹存储在文件 Q 中。80 亿个 64 位的语义指纹文件 F 大小是 64GB，压缩后小于 32GB。一批有 1MB 大小的语义指纹需要批量查询，因此假设文件 Q 的大小是 8MB。Google 把文件 F 和 Q 存放在 GFS 分布式文件系统中，文件分成多个 64MB 的组块，每个组块复制到一个集群中的 3 个随机选择的机器上，每个组块在本地系统存储成文件。

使用 MapReduce 框架，整个计算可以分成两个阶段。在第一阶段，有和 F 的组块数量一样多的计算任务（在 MapReduce 术语中，这样的任务叫做 mapper）。

每个任务以整个文件 Q 作为输入在某个 64MB 的组块上求解海明距离问题。

一个任务以发现的一个近似重复的语义指纹列表作为输出。在第二阶段，MapReduce 收集所有任务的输出，删除重复发现的语义指纹，产生一个唯一的、排好序的文件。

Google 用 200 个任务（mapper），扫描组块的合并速度在 1GB/s 以上。压缩版本的文件 Q 大小大约是 32GB（压缩前是 64GB），因此总的计算时间少于 100 秒。压缩对于速度的提升起了重要作用，因为对于固定数量的任务（mapper），时间大致正比于文件 Q 的大小。



## 5.5 中文关键词提取

关键词提取是文本信息处理的一项重要任务，例如可以利用关键词提取来发现新闻中的热点问题。和关键词类似，很多政府公文也有主题词描述。上下文相关广告系统也可能用到关键词提取技术。可以给网页自动生成关键词来辅助搜索引擎优化（SEO）。

有很多种方法可以应用于关键词提取，例如基于训练的方法和基于图结构挖掘的方法、基于语义的方法等。KEA (<http://www.nzdl.org/Kea>) 是一个开源的关键词提取项目。

### 5.5.1 关键词提取的基本方法

提取关键词整体流程如图 5-10 所示。



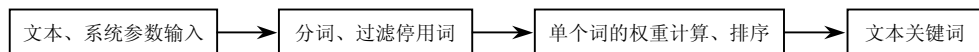


图 5-10 提取关键词调整流程

为了调节计算过程中用到的参数，可以建立关键词提取训练库。训练库包括训练文件（x.txt）和对应的关键词文件（x.key）。

依据以下几点来判断词的权重。

- 利用 TF\*IDF 公式，计算每个可能的关键词的 TF\*IDF：统计词频和词在所有文档中出现的总次数。TF（Term Frequency）代表词频，IDF（Invert Document Frequency）代表文档频率的倒数。比如说“的”在 100 文档中的 40 篇文档中出现过，则文档频率 DF（Document Frequency）是 40，IDF 是 1/40。“的”在第一篇文档中出现了 15 次，则  $TF*IDF（的）= 15 * 1/40=0.375$ 。另外一个词“反腐败”在这 100 篇文档中的 5 篇文档中出现过，则 DF 是 5，IDF 是 1/5。“反腐败”在第一篇文档中出现了 5 次，则  $TF*IDF（反腐败）= 5 * 1/5=1$ 。结果是： $TF*IDF（反腐败）> TF*IDF（的）$ 。
- 利用位置信息：开始和结束位置的词往往更可能是关键词。比如，利用下面的经验公式：

```
double position = t.startOffset() / content.length();
position = position * position - position + 2;
```

或者利用一个分段函数，首段或者末段的词的权重更大。

- 标题中出现的词比内容中的词往往更重要。
- 利用词性信息：关键词往往是名词或者名词结尾的词，而介词，副词，动词结尾的词一般不能组成词组。
- 利用词或者字的互信息： $I(X,Y)=\log_2 \frac{P(X,Y)}{P(X)P(Y)}$ 。互信息大的单字有可能是关键词。比如  $I(福,娃)=\log_2 \frac{P(福,娃)}{P(福)P(娃)}$ 。
- 利用标点符号：《 》和 “ ” 之间的文字更有可能是关键词。例如：“汉芯一号”造假案。
- 构建文本词网络：将单个词语当作词网络的节点，若两个词语共同出现在文本的一句话中，网络中对应节点建立一条权值为 1.0 的边。在文本词网络上运行图结构挖掘算法（例如 Page Rank 或 HITS 算法）对节点进行权值计算。

- 把出现的名词按语义聚类，然后提取出有概括性的词作为关键词。

首先定义词及其权重的描述类：

```
public class WordWeight implements Comparable<WordWeight> {
    public String word; //单词
    public double weight; //权重
    protected WordWeight(String word, double weight) {
        this.word = word;
        this.weight = weight;
    }
    public String toString() {
        return word + ":" + weight;
    }
    public int compareTo(WordWeight obj) {
        WordWeight that = obj;
        return (int) (that.weight - weight);
    }
}
```

返回关键词的主要代码如下所示：

```
//全部待选的关键词放入 PairingHeap 实现的优先队列
PairingHeap<WordWeight> h = new PairingHeap<WordWeight>();
//把单个单词放入优先队列
for (Entry<String,Double> it : wordTable.entrySet()) {
    word = it.getKey();
    java.lang.Double tempDouble;
    if( word.length() ==1)
        tempDouble = new Double(0.0);
    else
        tempDouble = it.getValue();
    h.insert( new WordWeight(word,tempDouble.doubleValue()) );
}
//把词组放入优先队列
for(WordWeight we : ngram) {
    h.insert(we);
}
retNum = Math.min(retNum,h.size()); //返回的关键词数量

WordWeight[] fullResults = new WordWeight[retNum]; //关键词返回结果
for(int i=0;i<retNum;++i) {
    fullResults[i] = (WordWeight)h.deleteMin();
}
```



其中为了返回权重最大的几个关键词，用到了 PairingHeap 实现的优先队列。

### 5.5.2 HITS 算法应用于关键词提取

首先介绍 HITS（Hypertext Induced Topic Selection）算法的原理与实现。HITS 算法可以选出有向带权重图中的最重要的节点。

每个节点有 Authority 和 Hub 两个值。Authority 值可以理解为该节点的权威性，也就是重要度。若 B 节点上有指向 A 节点的边，则称 B 为 A 的导入节点，这说明 B 认为 A 有指向价值，是一个“重要”的节点。所以 Authority 值是指向该节点的所有节点的 Hub 值之和。

如果一个节点指向另外一个节点，则可以看做这个节点向另外一个节点投了一票。Hub 值可以理解为该节点的投票可信度。Hub 值是该节点指向的那些节点的 Authority 值之和。可以看到，Authority 和 Hub 值以互相递归的方式定义。

HITS 算法执行一系列迭代过程，每个过程由如下两步组成——Authority 更新：更新每个节点的 Authority 值成为指向该节点的所有节点的 Hub 值之和；Hub 更新：更新每个节点的 Hub 值成为该节点指向的所有节点的 Authority 值之和。

用如下的算法计算一个节点的 Hub 值和 Authority 值。

- ① 设置每个节点的 Hub 值和 Authority 值为 1。
- ② 执行 Authority 更新规则。
- ③ 执行 Hub 更新规则。
- ④ 对 Hub 值和 Authority 值归一化。其中，Hub 值的归一化方式是每个 Hub 值除以所有的 Hub 值的平方和。Authority 值的归一化方式是每个 Authority 值除以所有的 Authority 值的平方和。
- ⑤ 重复 ② 直到达到指定的迭代次数，或者 Hub 值和 Authority 值的变化很小为止。

基本的算法实现代码如下所示：

```
public void computeHITS(int numIterations) {
    while(numIterations-->0 ) { //如果没有超过指定迭代次数
        for (int i = 1; i <= graph.numNodes(); i++) {
            //更新 Authority 值
        }
    }
}
```

```

        Map<Integer,Double> inlinks    = graph.inLinks(new
        Integer(i));
        double authorityScore = 0;

        for (Integer id:inlinks.keySet()) {
            authorityScore += (hubScores.get(id)).doubleValue();
        }
        authorityScores.put(new Integer(i), new Double
        (authorityScore));
    }
    for (int i = 1;i <= graph.numNodes(); i++)//更新 hub 值
    {
        Map<Integer,Double> outlinks  = graph.outLinks(new
        Integer(i));
        double hubScore = 0;
        for (Integer id:outlinks.keySet()) {
            hubScore += (authorityScores.get(id)).doubleValue();
        }

        hubScores.put(new Integer(i),new Double(hubScore));
    }
    normalize(authorityScores); //归一化 authority 值
    normalize(hubScores); //归一化 hub 值
}
}

```

如果认为节点之间连接的重要程度不一样，也就是指向关系有强弱之分，则考虑节点之间的连接重要度的实现。

```

public void computeWeightedHITS(int numIterations) {
    while(numIterations-->0 ) {
        for (int i = 1; i <= graph.numNodes(); i++) {
            Map<Integer,Double> inlinks    = graph.inLinks(new
            Integer(i));
            Map<Integer,Double> outlinks  = graph.outLinks(new
            Integer(i));
            double authorityScore = 0;
            double hubScore = 0;
            for (Entry<Integer,Double> in:inlinks.entrySet()) {
                authorityScore+=(hubScores.get(in.getKey())).double
                Value() * in. getValue();
            }

            for (Entry<Integer,Double> out:outlinks.entrySet()) {
                hubScore += (authorityScores.get(out.getKey())).

```



```
        doubleValue() * out.getValue());
    }

    authorityScores.put(new Integer(i), new Double(authorityScore));
    hubScores.put(new Integer(i), new Double(hubScore));
}
normalize(authorityScores);
normalize(hubScores);
}
```

为了确保引号里的词一定最重要，例如：“金正日”加了引号，所以这个词很重要。还有些词在文档的标题中出现，也很重要。所以要修改 HITS 算法，把节点的初始 Authority 值提高，并且保证它不会变得很低。例如，把用 TF\*IDF 方法计算的词权重作为节点的初始 Hub 值或 Authority 值。

### 5.5.3 从网页中提取关键词

从网页中提取关键词的处理流程是：

- ① 从网页中提取正文；
- ② 从正文中提取关键词。

在 H1 标签中的词，或者<B>黑体加粗的词可能更重要，更有可能是网页的关键词。另外，Meta 中的 KeyWords 描述也有可能真实的反映了该网页的关键词，例如：

```
<meta name="keywords" content="公判" />
```



## 5.6 相关搜索词

搜索引擎中往往有个可选搜索词的列表，当搜索结果太少的时候，可以帮助用户扩展搜索内容，或者当搜索结果过多的时候，可以帮助用户深入定向搜索。一种方法是从搜索日志中挖掘字面相似的词作为相关搜索词列表。从一个给定的词语挖掘多个相关搜索词，可以用编辑距离为主的方法查找一个词的字面相似词，如果候选的相关搜索词很多，就要筛选出最相关的 10 个词。

### 5.6.1 挖掘相关搜索词

下面是利用 Lucene 筛选给定词的最相关词的方法。

```
private static final String TEXT_FIELD = "text";
/**
 * @param words 候选相关词列表
 * @param word 要找相关搜索词的种子词
 * @return
 * @throws IOException
 * @throws ParseException
 */
static String[] filterRelated(HashSet<String> words, String word) {
    StringBuilder sb = new StringBuilder();
    for(int i=0;i<word.length();++i) {
        sb.append(word.charAt(i));
        sb.append(" ");
    }
    RAMDirectory store = new RAMDirectory();
    //按字生成索引和查找，也可以按细粒度的词分开
    IndexWriter writer = new IndexWriter(store, new Standard
    Analyzer(), true);
    for(String text:words) {
        Document document = new Document();
        Field textField = new Field(TEXT_FIELD, text,
                                   Field.Store.YES, Field.Index.TOKENIZED);
        document.add(textField);
        writer.addDocument(document);
    }
    writer.close();
    IndexSearcher searcher = new IndexSearcher(store);
    QueryParser queryParser = new QueryParser(TEXT_FIELD,
        new StandardAnalyzer());
    Query query = queryParser.parse(sb.toString());
    Hits hits = searcher.search(query);
    int maxRet = Math.min(10, hits.length());
    String[] relatedWords = new String[maxRet];
    for (int i = 0; i < maxRet ; i++) {
        Document document = hits.doc(i);
        String text = document.get(TEXT_FIELD);
        System.out.println(text);
        relatedWords[i]=text;
    }
    searcher.close();
    store.close();
}
```



```
        return relatedWords;  
    }  
}
```

上述代码整理出这样的相关词表，第一列是关键词，后续是 10 个以内的相关搜索词：

```
集福轩婚礼%集福轩  
手机定位跟踪系统%手机定位系统%手机定位%手机定位仪器  
喷绘材料卖店电话%我要喷绘材料卖店电话  
厦门房产%厦门租房%厦门新闻%厦门桑拿%房产%青岛房产%厦门%恒雄房产  
送水果%送水%水果  
三星传真机%三星手机
```

另外一种方法，可以把多个用户共同查询的词看成相关搜索词，需要有记录用户 IP 的搜索日志才能实现。

然后通过 RelatedEngine 类查找某个关键词的相关词。

```
RelatedEngine re =new RelatedEngine(  
    new File("D:/lg/work/xiaoxishu/dic/relatedwords.txt"));  
String word = "徐家汇";  
String[] relatedWords = re.getRelated(word);  
for(String w : relatedWords) {  
    System.out.println(w);  
}
```

上述代码的输出如下：

```
上海徐家汇  
徐汇  
徐家汇价格是  
上房徐家汇路附近有吗
```

当我们需要为新建立的搜索引擎开发相关搜索时，如果没有搜索日志而用户文本很多的时候，我们可以按以下步骤进行操作。

- ① 运行 IndexMaker 从待搜索的文档中提取关键词并生成索引。
- ② 运行 RelatedWords 从索引生成相关词表。

另外还考虑用日志记录用户对相关搜索词的选择。如果用户也选择了这个词，那搜索词肯定和这个选择词相关了。

隐含语义索引（Latent Semantic Indexing, LSI）的原理是在相同上下文中的词有相似的含义。<http://code.google.com/p/airhead-research/>是 Java 版本的实现。

### 5.6.2 使用多线程计算相关搜索词

如果要计算任意两个查询词之间的相关性，则发现相关搜索词的时间复杂度是  $O(n^2)$ 。例如要分析 10 兆多的相关搜索词的用时，则单线程的计算量可能长达 24 小时。

我们使用 Java 中自带的轻量级线程池来实现数据分析。JDK 1.5 以后的版本提供了一个轻量级线程池 `ThreadPool`。可以使用线程池执行一组任务，最简单的任务不返回值给主调线程。要返回值的任务可以实现 `Callable<T>` 接口，线程池执行任务并通过 `Future<T>` 的实例获取返回值。

实现 `Callable` 方法任务类的主要实现代码如下所示：

```
public class FindSimCall implements Callable<String[]> {
    private HashSet<String> words; //总的搜索词集合
    private String s;              //待发现相关词的词

    public FindSimCall(HashSet<String> w ,String source) {
        words = w;
        s = source;
    }

    @Override
    public String[] call() throws Exception {
        System.out.println(s);
        //形成 related words 列表
        return relatedWords;
    }
}
```

主线程类的实现代码如下所示：

```
int threads = 4;
ExecutorService es = Executors.newFixedThreadPool(threads);

Set<Future<String[]>> set = new HashSet<Future<String[]>>();
```





```
for (final String s : words) {
    FindSimCall task = new FindSimCall(words,s);
    Future<String[]> future = es.submit(task);
    set.add(future);
}

FileOutputStream fos = new FileOutputStream(relatedWordsFile);
OutputStreamWriter osw = new OutputStreamWriter(fos,"GBK");
BufferedWriter writer = new BufferedWriter(osw);

for (Future<String[]> future : set) {
    String[] ret = future.get();

    for(String word:ret) {
        writer.write("%"+word);
    }
    writer.write( "\r\n" );
}
writer.close();
```

采用线程池可以充分利用多核 CPU 的计算能力，并且简化了多线程的实现。



## 5.7 信息提取

据说，看同一段视频，聪明人和一般人的差别在于：他会从视频中提取出自己感兴趣的信息。本节只介绍文本信息提取。

从文本中抽取用户感兴趣的事件、实体和关系，被抽取出来的信息以结构化的形式描述，然后存储在数据库中，为各种应用提供服务。。

例如：从新闻报道中抽取恐怖事件的详细情况，包括时间、地点、作案者、受害者、袭击目标、使用的武器等；从经济新闻中抽取公司发布新产品的情况，公司名、产品名、发布时间、产品性能等。

华盛顿大学开发的开放式互联网信息提取系统“TEXTRUNNER”提取实体和它们之间的关系。例如“海德公园”和“英国”的关系是“位于”。当用户提问“海德公园位于哪里？”时，系统可以根据提取出的信息回答“英国”。依赖手工编写的提取规则，或者手工标注的训练例子来实现信息提取（Information Extraction）。

GATE (<http://gate.ac.uk>) 是一个应用广泛的信息抽取的开放型基础架构，该系统对语言处理的各个环节——从语料收集、标注、重用到系统评价均能提供很好的支持。我们这里实现一个简化版本的信息抽取系统。

输入“北京盈智星公司”切分后标注成“北京/行政区划 盈智星/关键词 公司/功能词”。根据标注的结果可以提取出“盈智星”这样的关键词。有个基本的词典用来存放行政区划、功能词表等特征，例如“北京”这个词是在一个行政区划词表中，“公司”在另外一个功能词表中。提取地址会碰到很多词典中没有的需要识别的未登录词，例如“高东镇高东二路”，需要把“高东二路”这样不在词典中的路名识别出来。可以先把输入串抽象成待识别的序列“镇后缀 UNKNOWN 号码 街后缀”，然后利用规则（也叫模板）来识别并提取信息。未登录地址识别规则可以表示成如下形式：

镇后缀 未登录街道 =>镇后缀 UNKNOWN 号码 街后缀

转换成代码实现如下所示：

```
lhs = new ArrayList<AddressSpan>(); //左边的符号
rhs = new ArrayList<AddressType>(); //右边的符号
//镇后缀 UNKNOWN 号码 街后缀
rhs.add(AddressType.SuffixTown);
rhs.add(AddressType.Unknown);
rhs.add(AddressType.No);
rhs.add(AddressType.SuffixStreet);
//镇后缀 未登录街道
lhs.add(new AddressSpan(1,AddressType.SuffixTown)); //归约长度是 1
//把“UNKNOWN 号码 街后缀”3个符号替换成“未登录街道”，因此归约长度是 3
lhs.add(new AddressSpan(3,AddressType.Street));
//加到规则库
addProduct(rhs, lhs);
```

“UNKNOWN 号码 街后缀”合并成“街道”，可以记录下内部结构，这样方便后续处理。

为了提高提取的准确性，规则往往设计成比较长的形式。长的规则往往更多参考上下文，覆盖面小，但是更准确。短的规则会影响更多的提取结果，可能这一条信息靠这条规则提取正确了，却有更多的其他记录受影响。

设计规则存储格式为“右边的符号列表@左边的符号列表”。左边的符号列表用“整数 值，类型”来表示，例如：



```
Town,SuffixProvince @ 2, Province
Unknow,SuffixTown,Unknow,No,SuffixStreet@2,Town,3,Street
Town,Unknow,No,SuffixStreet@1,Town,3,Street
City,Unknow,Street@1,City,2,Street
Unknow,SuffixStreet@2,Street
Unknow,SuffixLandMark@2,LandMark
Unknow,No,SuffixStreet@3,Street
```

读入规则的程序如下所示：

```
StringTokenizer st = new StringTokenizer(line, "@");//分成左右字符串
StringTokenizer rhst = new StringTokenizer(st.nextToken(), ",");
//逗号分隔
StringTokenizer lhst = new StringTokenizer(st.nextToken(), ",");
//逗号分隔
ArrayList<PoiSpan> rhs = new ArrayList<PoiSpan>(); //右边的符号
ArrayList<PoiType> lhs = new ArrayList<PoiType>(); //左边的符号
while (rhst.hasMoreTokens()) {
    lhs.add(PoiType.valueOf(rhst.nextToken())); //左边类型
}
while (lhst.hasMoreTokens()) {
    rhs.add(new PoiSpan(Integer.parseInt(lhst.nextToken()),
//右边符号长度
                                PoiType.valueOf(lhst.nextToken()))); //右边符号类型
}
addProduct(lhs, rhs); //加入到规则库
```

可以从 Java 源代码中抽取出规则到配置文件中，实现代码如下所示：

```
String[] sa=str.split("addProduct");//分割每一条规则
for(String s : sa){ //遍历每一条规则并处理
    //通过正则表达式匹配找出该条规则中所有的右边的符号
    Pattern p=Pattern.compile("rhs\\\\.add\\.POIType1\\.\\.([a-zA-Z]+)");
    Matcher m=p.matcher(s);
    String result="";
    while(m.find()) {
        result=result+m.group(1)+",";
    }
    if("").equals(result)) {
        break;
    }

    //通过正则表达式匹配找出该条规则中所有的左边的符号
    result=result.substring(0, result.length()-1)+"@";
```

```

Pattern p2=Pattern.compile("lhs\\.add.new\\sPoiSpan1\\.([0-9]+),
\\sPOIType1\\.([a-zA-Z]+)");
Matcher m2=p2.matcher(s);
while(m2.find()) {
    String num=m2.group(1);
    String type=m2.group(2);
    result+=num+","+type+",";
}
//输出提取出的规则
System.out.println(result.substring(0, result.length()-1));
}

```

规则替换可能会进入死循环，因为可能出现重复应用规则的情况。如果规则的左边部分小于右边部分，也就是说替换后的长度越来越短，应用这样的规则不会导致死循环。当规则的左边部分和右边部分相等时，可以用 **Token** 类型的权重和来衡量，规则左边部分的权重和必须小于右边部分的权重和。这样的规则让应用于匹配序列的 **Token** 类型的权重和越来越小，所以也不会产生死循环。使用 **ordinal** 方法取得的枚举类型的内部值大小作为权重。下面是检查规则的实现方法：

```

/**
 * 规则校验
 *
 * @return true 表示规则不符合规范 false 表示符合符合规范
 */
public boolean check(ArrayList<DocType> key, ArrayList<DocSpan> lhs) {
    boolean isEqual = false;
    for (DocSpan span : lhs) {
        if (span.length > 1) {
            return isEqual;
        }
    }
    int leftCount = 0;
    int rightCount = 0;
    for (DocType dy : key) {
        leftCount += dy.ordinal();
    }
    for (DocSpan sd : lhs) {
        rightCount += sd.type.ordinal();
    }

    if (leftCount <= rightCount) {
        isEqual = true;
    }
}

```



```
return isEqual;
}
```

规则很多的时候，需要看一段文本匹配上了哪一条规则，或者考查某一条具体的规则可能产生的影响，先总体执行一遍数据，然后看哪些数据用了这条规则。

信息提取的流程说明如下。

- ① 定义词的类别。
- ② 根据词库做全切分。
- ③ 最大概率动态规划求解。
- ④ HMM 词性标注。
- ⑤ 基于规则的未登录词识别。
- ⑥ 根据切分和标注的结果提取信息。

例如，为农业相关的文档提取出作物名称、对应季节、适用地区等信息。

```
public enum DocType {
    Product, //作物名称
    Pronoun, //代词
    Address, //地名
    Season, //季节
    Start, //虚拟类型，开始状态
    End //虚拟类型，结束状态
}
```

可以自己建几个简单的词表，例如季节词表 `season.txt`，存放内容如：

春  
夏  
秋  
冬

作物名称有个 `product.txt` 词表，存放内容如：

大豆  
高粱

然后通过 DicDoc 类加载这些词，代码如下所示：

```
private DicDoc() {
    //加载字典
    load("product.txt", DocType.Product); //农作物
    load("address.txt", DocType.Address); //地址
    load("season.txt", DocType.Season); //季节
}
```

信息提取的关键在于定义相关规则，用户定义好规则后，程序会按照指定的规则提取相关信息，规则越多，提取的信息越精确。另外，可以把需要优先匹配的规则放到前面，因为规则库中放在前面的规则会先匹配上。



## 5.8 拼写检查与建议

输错电话号码，往往只是得到简单的提示“没有这个电话号码”。但是在搜索框中输入错误的搜索词，搜索引擎往往会显示“您是不是要找”以提示这个正确的词，这个功能也叫做“Did you mean”。

拼写检查是查询处理极为重要的一个组成部分。在网络搜索引擎用户提交的查询中有大约 10%到 15%的拼写错误，拼写检查就是对错误的词给出正确的提示。如果有个正确的词和用户输入的词很近似，则用户的输入可能是错误的。

```
IndexSearcher is = new IndexSearcher(originalIndexDirectory);
Hits hits = is.search(query); //原始查询

String suggestedQueryString = null; //提示查询

//如果搜索返回结果的数量小于一个阈值
//或者匹配第一个结果的分值小于最小值就查找提示词
if (hits.length() < minimumHits || hits.score(0) < minimumScore) {
    Query didYouMean = didYouMeanParser.suggest(queryString);
    //调用拼写检查算法
    if (didYouMean != null) {
        suggestedQueryString = didYouMean.toString(defaultField);
    }
}
```

如果返回结果的数量很少，可以直接把提示词的搜索结果放在原查询词返回结果的下面。



查询日志中包含大量的简单错误的例子，如下所示：

```
poiner sisters -> pointer sisters
brimingham news -> birmingham news
ctamarn sailing -> catamaran sailing
```

类似这些错误可以通过建立正误词表来检查。然而，除此之外，将有许多查询日志包含与网站、产品、公司相关的词，对于这样的开放类的词不可能在标准的拼写词典中发现。以下是来自同一个查询日志的一些例子：

```
akia 1080i manunal -> akia 1080i manual
ultimatwarcade -> ultimatearcade
mainsourcebank -> mainsource bank
```

因此不存在万能的词表，垂直（网站）搜索引擎往往需要整理和自己行业（网站）相关的词库才能达到好的匹配效果。可以从搜索日志中挖掘出“错误词->正确词”这样的词对，例如“飞利浦->飞利浦”。

根据正误词表替换用户输入，具体代码如下所示。

```
public static String replace(String content) {
    int len = content.length();
    StringBuilder ret = new StringBuilder(len);
    ErrorDic.PrefixRet matchRet = new ErrorDic.PrefixRet(null, null);

    for(int i=0; i<len;){
        errorDic.checkPrefix(content, i, matchRet); //检查是否存在从当前
        //位置开始的错词
        if(matchRet.value == ErrorDic.Prefix.Match) {
            ret.append(matchRet.data);
            i=matchRet.next; //下一个匹配位置
        }
        else //从下一个字符开始匹配
        {
            ret.append(content.charAt(i));
            ++i;
        }
    }
    return ret.toString();
}
```

因为在各种语言中导致用户输入错误的原因不一样，所以每种语言的正误词对的挖掘

方式各有不同。对英文单词的搜索需要专门针对英文的拼写检查，对中文词的搜索需要专门针对中文的拼写检查。

为了讨论对搜索引擎查询最有效的拼写检查技术，我们首先看一下如何对一般的文本进行拼写检查。对于中文文本拼写检查，可以考虑“黑马校对软件”；对于英文的文档，有 Aspell (<http://aspell.net/>) 拼写帮助。

$$\text{Spell}(w) = \arg \max_{c \in C} P(c | w) = \arg \max_{c \in C} \frac{P(w | c)P(c)}{P(w)}$$

对于任何  $c$  来讲，出现  $w$  的概率  $P(w)$  都是一样的，从而我们在上式中忽略它，写成：

$$\text{Spell}(w) = \arg \max_{c \in C} P(w | c)P(c)$$

这个式子有三个部分，从右到左分别介绍如下。

- $P(c)$ : 文章中出现一个正确拼写词  $c$  的概率，也就是说，在英语文章中， $c$  出现的概率有多大呢？因为这个概率完全由英语这种语言决定，我们称之为语言模型。好比说，英语中出现 **the** 的概率  $P(\text{'the'})$  就相对较高，而出现  $P(\text{'zxzxzxzyy'})$  的概率接近 0 (假设后者也是一个词的话)。
- $P(w|c)$ : 在用户想键入  $c$  的情况下敲成  $w$  的概率。因为这个代表用户会以多大的概率把  $c$  敲错成  $w$ ，因此被称为误差模型。
- $\arg \max c$ : 用来枚举所有可能的  $c$  并且选取概率最大的，因为我们有理由相信，一个（正确的）单词出现的频率较高，用户又容易把它敲成另一个错误的单词，那么，敲错的单词应该被更正为这个正确的。

为什么把最简单的一个  $P(c|w)$  变成两项复杂的式子来计算呢？因为  $P(c|w)$  就是和这两项同时相关的，因此拆成两项反而容易处理。举个例子，比如一个单词 **thew** 拼错了。看上去 **thaw** 应该是正确的，因为就是把 **a** 敲成 **e** 了。然而，也有可能用户想要的是 **the**，因为 **the** 是英语中常见的一个词，并且很有可能打字的时候手不小心从 **e** 滑到 **w**。因此，在这种情况下，我们想要计算  $P(c|w)$ ，就必须同时考虑  $c$  出现的概率和从  $c$  到  $w$  的概率。把一项拆成两项让这个问题更加容易更加清晰。

对于给定词  $w$  可以通过编辑距离挑选出相似的候选正确词  $c$  的集合。编辑距离越小，候选正确词越少，计算也越快。76% 的正确词和错误词的编辑距离是 1，还需要考虑编辑距





离是 2 的情况，99% 的正确词和错误词的编辑距离在 2 以内。因此对于拼写检查来说，找出编辑距离在 2 以内的候选正确词  $c$  的集合就可以了。这是一个模糊匹配的问题。

### 5.8.1 模糊匹配问题

如何从一个大的正确词表中找出和输入词编辑距离小于  $k$  的词集合？逐条比较编辑距离太慢。

编辑距离自动机的基本想法是：构建一个有限状态自动机准确地识别出和目标词在给定的编辑距离内的字符串集合。可以输入任何词，然后自动机可以基于是否和目标词的编辑距离最多不超过给定距离从而接收或拒绝它。由于 FSA 的内在特性，上述过程可以在  $O(n)$  时间内判断是可以接收或应该拒绝。这里， $n$  是测试字符串的长度。而标准的动态规划编辑距离计算方法需要  $O(m*n)$  时间，这里  $m$  和  $n$  是两个输入单词的长度。因此编辑距离自动机可以更快地检查许多单词和一个目标词是否在给定的在最大距离内。

单词 “food” 的编辑距离自动机形成的非确定有限状态自动机，最大编辑距离是 2。开始状态在左下，状态使用  $n^e$  标记风格命名。这里  $n$  是目前为止正确匹配的字符数， $e$  是错误数量。垂直转换表示未修改的字符，水平转换表示插入，两类对角线转换表示替换（用 \* 标记的转换）和删除（空转换）。

单词 “food” 的长度是 4，所以图 5-11 中有 5 列。允许两次错误，所以有 3 行。

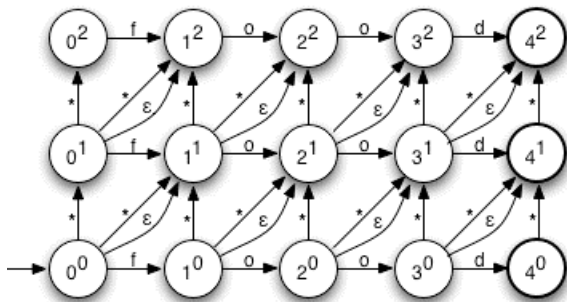


图 5-11 编辑距离自动机

编辑距离自动机的实现代码如下所示：

```
public static NFA levenshteinAutomata(String term, int k) {  
    NFA nfa = new NFA(new State(0, 0)); // 根据初始状态构建非确定有限状态机  
  
    for (int i = 0; i < term.length(); ++i) {
```

```

char c = term.charAt(i);
for (int e = 0; e < (k + 1); ++e) {
    //正确字符
    nfa.addTransition(new State(i, e), c, new State(i + 1, e));
    if (e < k) {
        //删除
        nfa.addTransition(new State(i, e), NFA.ANY, new State(i,
            e + 1));
        //插入
        nfa.addTransition(new State(i, e), NFA.EPSILON,
            new State(i + 1, e + 1));
        //替换
        nfa.addTransition(new State(i, e), NFA.ANY, new State(i
            + 1, e + 1));
    }
}
}
for (int e = 0; e < (k + 1); ++e) {
    if (e < k)
        nfa.addTransition(new State(term.length(), e), NFA.ANY,
            new State(term.length(), e + 1));
    nfa.addFinalState(new State(term.length(), e)); //设置结束状态
}
return nfa;
}

```

看某个单词是否和给定单词相似:

```

//构建编辑距离自动机
NFA levenshteinAutomata = NFA.levenshteinAutomata("food", 1);
//根据幂集构造转换成确定有限状态机
DFA dfa = levenshteinAutomata.toDFA();
//看单词 foxd 是否能够被接收
System.out.println(dfa.accept("foxd"));

```

把正确的词典和条件 (Levenshtein automata) 都表示成 DFA, 有可能高效地对两个 DFA 取交集 (intersect), 从词典中找到满足条件的词。步调一致地遍历两个 DFA, 仅跟踪两个 DFA 共有的边, 并且记录走过来的路径。任何时候两个 DFA 都在结束状态, 输出词典 DFA 对应的单词。

```

public static ArrayList<String> intersect(DFA dfa1, DFA dfa2) {
    ArrayList<String> match = new ArrayList<String>(); //找到的正确单词集合
    Stack<StackValue> stack = new Stack<StackValue>();
    stack.add(new StackValue("", dfa1.startState, dfa2.startState));
}

```



```
while (!stack.isEmpty()) {
    StackValue stackValue = stack.pop();
    Set<Character> ret =
        intersection(dfa1.edges(stackValue.s1), dfa2.edges
            (stackValue.s2));
    for(char edge:ret) {
        State state1 = dfa1.next(stackValue.s1, edge);
        State state2 = dfa2.next(stackValue.s2, edge);
        if(state1!=null&& state2!=null){
            stackValue.s += edge;
            stack.add(new StackValue(stackValue.s, state1, state2));
            if(dfa1.isFinal(state1) && dfa2.isFinal(state2)){
                match.add(stackValue.s);
            }
        }
    }
}
return match;
}
```

可以使用标准 Trie 树代替 DFA，标准 Trie 树中存储了正确词库，使用方法如下所示：

```
//错误词
NFA lev = NFA.levenshteinAutomata("foo",1);
DFA dfa = lev.toDFA();

//正确词表
Trie<String> stringTrie = new Trie<String>();
stringTrie.add("food", "food");
stringTrie.add("hammer", "hammer");
stringTrie.add("hammock", "hammock");
stringTrie.add("ipod", "ipod");
stringTrie.add("iphone", "iphone");
//返回相似的正确的词
ArrayList<String> match = DFA.intersect(dfa, stringTrie);
```

## 5.8.2 英文拼写检查

对英文报关公司名 101919 条进行统计，有拼写错误的为 16663 条，出错概率为 16.35%。大部分是正确的，如果所有的词都在正确词表中，则不必再查找错误，否则先检查错误词表，最后仍然不确定的，提交查询给搜索引擎，看看是否有错误。

正确词的词典格式中每行一个词，包括词本身和词频，样例如下所示：

```

biogeochemistry : 1
repairer : 3
wastefulness : 3
battier : 2
awl : 3
preadapts : 1
surprisingly : 3
stuffiest : 3

```

因为互联网中的新词不断出现，正确的词并不是来源于固定的词典，而是来源于搜索的文本本身。下面直接从文本内容提取英文单词，不从索引库中提取的原因是 Term 可能经过词干化处理了，所以我们用 StandardAnalysis 再次处理。

```

java.io.StringReader input = new java.io.StringReader(content);
TokenStream tokenizer = new StandardTokenizer(input);
for (Token t = tokenizer.next(); t != null; t = tokenizer.next())
{
    if( isAllLetter(t.termText()) &&
        (t.termText().length()>=3) &&
        (t.termText().length()<=30) )
    {
        System.out.println(t.termText());
        fpSource.write(t.termText().toLowerCase());
        fpSource.write(" : 1\n");
    }
}

```

可以根据发音计算用户输入词和正确词表的相似度，还可以根据字面的相似度来判断是否输入错误，并给出正确的单词提示。也可以参考一下开源的 Spell Checker 的实现，例如 Aspell (<http://aspell.net/>)。

特别要考虑公司名中的拼写错误。一般首字母拼写错误的可能性很小。可以简单地先对名称排序，然后再比较前后两个公司名就可以检测出一些非常相似的公司名称了。

另外可以考虑抓取 Google 的拼写检查结果。

```

public static String getGoogleSuggest(String name) throws Exception
{
    String searchWord = URLEncoder.encode(name,"utf-8");
    String searchURL = "http://www.google.com/search?q=" + searchWord;
    String strPages=DownloadPage.downloadPage(searchURL);//下载页面

```



```
String suggestWord="";
Parser parser=new Parser(strPages); //使用 HTMLParser 解析 html
NodeFilter filter=new AndFilter(new TagNameFilter("a"),new
HasAttributeFilter("class","spell"));
//取得符合条件的第一个节点
NodeList nodelist=parser.extractAllNodesThatMatch(filter);
int listCount=nodelist.size();

if(listCount>0)
{
    TagNode node=(TagNode)nodelist.elementAt(0);
    if (node instanceof LinkTag)//<a> 标签
    {
        LinkTag link = (LinkTag) node;
        suggestWord = link.getLinkText();//链接文字
    }
}

return suggestWord;
}
```

### 5.8.3 中文拼写检查

和英文拼写检查不一样，中文的用户输入的搜索词串的长度更短，从错误的词猜测可能的正确输入更加困难。这时候需要更多的借助正误词表，词典文本格式如下。

代款：贷款

阿地达是：阿迪达斯

诺基压：诺基亚

飞利浦：飞利浦

寂么沙洲冷：寂寞沙洲冷

欧米加：欧米茄

欧米枷：欧米茄

爱力信：爱立信

西铁成：西铁城

瑞新：瑞星

登心绒：灯心绒

前面一个词是错误的词条，后面是对应的错误词条。为了方便维护，我们还可以把这个词库存放在数据库中：

```
CREATE TABLE CommonMisspellings (
    [misword] [varchar] (50) COLLATE Chinese_PRC_CI_AS NULL ,
    --错误词
    [rightword] [varchar] (50) COLLATE Chinese_PRC_CI_AS NULL --正确词
)
```

除了人工整理，还可以从搜索日志中挖掘相似字串来找出一些可能的正误词对。比较常用的方法是采用编辑距离（Levenshtein Distance）来衡量两个字符串是否相似。编辑距离就是用来计算从原串（ $s$ ）转换到目标串（ $t$ ）所需要的最少的插入，删除和替换的数目。例如源串是“诺基压”，目标串是“诺基亚”，则编辑距离是1。

日志中有这样的记录：

```
2007-05-24 00:41:41.0781|DEBUG|221.221.167.147||喀尔喀蒙古|2
...
2007-05-24 00:43:45.7031|DEBUG|221.221.167.147||喀爾喀蒙古|0
...
```

假设有搜索结果返回的是正确词，无搜索返回的是错误词。挖掘日志的程序如下所示：

```
//存放挖掘的词及搜索出的结果数
HashMap<String,Integer> searchWords = new HashMap<String,Integer>();
while((readline=br.readLine())!=null) {
    StringTokenizer st = new StringTokenizer(readline,"|");
    if(!st.hasMoreTokens()) continue;
    st.nextToken();
    if(!st.hasMoreTokens())continue;
    st.nextToken();
    if(!st.hasMoreTokens())continue;
    st.nextToken();
    if(!st.hasMoreTokens())continue;
    st.nextToken();
    if(!st.hasMoreTokens())continue;
    st.nextToken();
    if(!st.hasMoreTokens())continue;
    //存放搜索词
    String key = st.nextToken();
    if(key.indexOf(":")>=0) {
        continue;
    }
    //如果已经处理过这个词就不再处理
    if(searchWords.containsKey(key)) {
```



```
        continue;
    }
    if(!st.hasMoreTokens())
    {
        continue;
    }
    String results = st.nextToken();
    int resultCount = Integer.parseInt(results);
    //得到搜索出的结果数

    for(Entry<String,Integer> e : searchWords.entrySet()) {
        int diff= Distance.LD(key, e.getKey()) ;
        if(diff ==1 && key.length()>2) {
            if( resultCount == 0 && e.getValue()>0 ) {
                // e.getKey() 是正确词，key 是错误词
                System.out.println(key +":"+ e.getKey());
                bw.write(key +":"+ e.getKey()+"\r\n");
            }
            else if(e.getValue()==0 && resultCount>0) {
                // key 是正确词，e.getKey() 是错误词
                System.out.println(e.getKey() +":"+ key);
                bw.write(e.getKey() +":"+ key+"\r\n");
            }
        }
    }
    searchWords.put(key, resultCount); //存放当前词及搜索出
    //的结果数
}
```

可以挖掘出如下一些错误与正确的词对。

瑜伽服：瑜伽服

落丽塔：洛丽塔

巴甫洛：巴甫洛夫

hello kiitty: hello kitty

...

除了根据搜索日志挖掘正误词表，还可以根据拼音或字形来挖掘。例如根据拼音挖掘出“周杰论：周杰伦”，根据字形挖掘出“浙江移动：浙江移动”。



## 5.9 自动摘要

减少原文的长度而保留文章的主要意思叫做摘要。摘要有各种形式。和搜索关键词相关的摘要，叫做动态摘要；只和文本内容相关的摘要叫做静态摘要。搜索引擎中显示的搜索结果就是关键词相关摘要的例子。按照信息来源来分有来源于单个文档的摘要和合并多个相关文档意思的摘要。单文档摘要精简一篇文章的主要意思，多文档摘要同时可以过滤掉出现在多篇文档中的重复内容。

除了用于搜索结果中显示的摘要，文本自动摘要还可以用于手机 WAP 网站显示摘要信息，还可以用于发送短信。

### 5.9.1 自动摘要技术

摘要的实现方法有摘取性的和概括性的。摘取性的方法相对容易实现，通常的实现方法是摘取文章中的主要句子。

MEAD (<http://www.summarization.com/mead/>) 是一个功能完善的多文档摘要软件，不过是 Perl 实现的。Classifier4J (<http://classifier4j.sourceforge.net/>) 包含一个简单的文本摘要实现，方法是抽取指定文本中的重要句子形成摘要。使用它的例子如下所示：

```
String input = "Classifier4J is a java package for working with text.  
Classifier4J includes a summariser.";  
//输入文章内容及摘要中需要返回的句子个数  
String result = summariser.summarise(input, 1);
```

上述代码的返回结果是：“Classifier4J is a java package for working with text.”。

### 5.9.2 自动摘要的设计

自动摘要的主要方法有基于句子重要度的方法和基于篇章结构的方法。前者相对成熟，后者还处于研究阶段。

Classifier4J 也是采用了句子重要度计算的简化方法。Classifier4J 通过统计高频词和句子分析来实现自动摘要。其主要流程说明如下。

- ① 取得高频词。





- ② 把内容拆分成句子。
- ③ 取得包含高频词的前  $k$  个句子。
- ④ 将句子按照在文中出现的顺序重新排列，添加适当的分隔符后输出。

统计文本中最常出现的  $k$  个高频词的基本方法说明如下。

- ① 在遍历整个单词序列时，使用一个散列表记录所有的单词频率。散列表的关键字是词，而值是词频，该过程花费  $O(n)$  时间。
- ② 对散列表按值从大到小排序。使用通常的排序算法花费  $O(n * \lg(n))$  时间。
- ③ 排序后，取前  $k$  个词。

为了优化②和③，可以不对散列表全排序，直接取前  $k$  个值最大的词。用到的方法是从数组中快速地选取最大的  $k$  个数。

快速排序基于分而治之（divide and conquer）的策略。数组  $A[p..r]$  被划分为两个（可能空）子数组  $A[p..q-1]$  和  $A[q+1..r]$ ，使得  $A[p..q-1]$  中的每个元素都小于等于  $A(q)$ ，而且小于等于  $A[q+1..r]$  中的元素。这里  $A(q)$  称为中值。可以根据快速排序的原理设计接口如下所示：

```
//根据随机选择的中值来选取最大的 k 个数，输入参数说明如下：
//a 待选取的数组
//size 数组的长度
//k 前 k 个值最大的词
//offset 偏移量
selectRandom(ArrayList<WordFreq> a, int size, int k, int offset)
```

根据快速排序的原理，实现选取最大的  $k$  个数的方法如下所示：

```
//把数组中的两个元素交换位置
public static <E extends Comparable<? super E>> void swap(ArrayList<E>
a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
static void selectRandom(ArrayList<WordFreq> a, int size, int k, int
offset) {
    if (size < 5) { //采用简单的冒泡排序方法对长度小于 5 的数组排序
```

```

        for (int i = offset; i < (size + offset); i++)
            for (int j = i + 1; j < (size + offset); j++)
                if (a.get(j).compareTo(a.get(i)) < 0)
                    swap(a, i, j);
        return;
    }
    Random rand = new Random();//随机选取一个元素作为中值
    int pivotIdx = partition(a, size, rand.nextInt(size) + offset,
offset);
    if (k != pivotIdx) {
        if (k < pivotIdx) {
            selectRandom(a, pivotIdx - offset, k, offset);
        } else {
            selectRandom(a, size - pivotIdx - 1 + offset, k, pivotIdx
+ 1);
        }
    }
}
static int partition(ArrayList<WordFreq> a, int size, int pivot, int
offset) {
    WordFreq pivotValue = a.get(pivot); //取得中值
    swap(a, pivot, size - 1 + offset);
    int storePos = offset;
    for (int loadPos = offset; loadPos < (size - 1 + offset); loadPos++){
        if (a.get(loadPos).compareTo(pivotValue) < 0) {
            swap(a, loadPos, storePos);
            storePos++;
        }
    }
    swap(a, storePos, size - 1 + offset);
    return (storePos);
}

```

参考 Classifier4J 的实现方法，中文自动摘要的基本实现方法包含如下 5 个步骤。

- ① 通过中文分词，统计词频和词性等信息，抽取出关键词。
- ② 把文章划分成一个个的句子。
- ③ 通过各句中关键词出现的情况定义出句子的重要度。
- ④ 确定前  $K$  个最重要的句子为文摘句。
- ⑤ 把文摘句按照在原文中出现的顺序输出成摘要。



文本摘要的主体程序如下所示：

```
ArrayList<CnToken> pItem = Tagger.getFormatSegResult(content); //分词
//关键词及对应的权重
HashMap<String,Integer> keyWords = new HashMap<String,Integer>(10);
WordCounter wordCounter = new WordCounter(); //词频统计
for (int i = 0; i < pItem.size(); ++i) {
    CnToken t = pItem.get(i);
    if (t.type().startsWith("n")) {
        wordCounter.addNWord(t.termText()); //增加名词词频
    } else if (t.type().startsWith("v")) {
        wordCounter.addVWord(t.termText()); //增加动词词频
    }
}
//取得出现的频率最高的五个名词
WordFreq[] topNWords = wordCounter.getWords(wordCounter.wordNCount);
for (int i = 0; i < topNWords.length; i++) {
    keyWords.put(topNWords[i].word, topNWords[i].freq);
}
//取得出现的频率最高的五个动词
WordFreq[] topVWords = wordCounter.getWords(wordCounter.wordVCount);
for (int i = 0; i < topVWords.length; i++) {
    keyWords.put(topVWords[i].word, topVWords[i].freq);
}
//把内容分割成句子
ArrayList<SentenceScore> sentenceArray = getSentences(content, pItem);
//计算每个句子的权重
for (SentenceScore sc : sentenceArray) {
    sc.score = 1;

    for (Entry<String,Integer> e : keyWords.entrySet()) {
        String word = e.getKey();
        if (sc.containWord(word)) {
            sc.score = sc.score * e.getValue();
        }
    }
}

//取得权重最大的三个句子
int minSize = Math.min(sentenceArray.size(), 3);
Select.selectRandom(sentenceArray, sentenceArray.size(), minSize, 0);
SentenceScore[] orderSen = new SentenceScore[minSize];
for (int i = 0; i < minSize; ++i) {
    orderSen[i] = sentenceArray.get(i);
}
```

```
//按句子在原文中出现的顺序输出
Arrays.sort(orderSen, posCompare); //句子数组按在文档中的位置排序
String summary = "";
for (int i = 0; i < minSize; i++) {
    String curSen = orderSen[i].getSentence(content);
    summary = summary.concat(curSen);
}
return summary;
```

这只是一个简单的文本摘要程序，优化的方法说明如下。

- 除了通过关键词，还可以通过提取基本要素（Basic Elements）来确定句子的重要程度。基本要素通过三元组<中心词，修饰，关系>来描述，其中中心词为该三元组的主要部分。
- 在提取关键词阶段，可以去掉停用词表，然后再统计关键词。也可以考虑利用同义词信息更准确的统计词频。
- 在划分句子阶段，可以记录句子在段落中出现的位置，在段落开始或结束出现的句子更有可能是关键句。同时可以考虑句型，陈述句比疑问句或感叹句更有可能是关键句。

首先定义句子类型：

```
public static enum SentenceType {
    declare, //陈述句
    question, //疑问句
    exclamation //感叹句
}
```

句子权重统计阶段考虑用句型来打分。

```
//判断句子类型
if(sc.type == SentenceScore.SentenceType.question) {
    sc.score *= 0.1;
}
else if(sc.type == SentenceScore.SentenceType.exclamation) {
    sc.score *= 0.5;
}
```

为了使输出的摘要意义连续性更好，有必要划分段落。识别自然段和更大的意义段。自然段一般段首缩进两个或四个空格。



在对句子打分时，除了关键词，还可以查看事先编制好的线索词表。表示线索词的权值，有正面的和负面的两种。文摘正线索词就是类似“总而言之”、“总之”、“本文”、“综上所述”等词汇，含有这些词的句子权重有加分。文摘负线索词可以是“比如”，“例如”等。如果句中包括这些词权重就会降低。

为了减少文摘句之间的冗余度，可以通过句子相似度计算减少冗余句子。具体过程说明如下。

- ① 将句子按其重要度从高到低排序。
- ② 抽取重要度最高的句子  $S_i$ 。
- ③ 选取候选句  $S_i$  后，调整剩下的每个待选句的重要度。待选句  $S_j$  的重要度按如下公式进行调整：

$$\text{Score}(S_j) = \text{Score}(S_j) - \text{sim}(S_i, S_j) * \text{Score}(S_i)$$

其中  $\text{sim}(S_i, S_j)$  是句子  $S_i$  和  $S_j$  的相似度。

- ④ 剩下的句子按重要度从高到低排序，选取重要度高的句子。
- ⑤ 重复 ③、④，直至摘要足够长为止。

最后为了输出的摘要通顺，还需要处理句子间的关联关系。例如下面的关联句子：

“这个节目，需要的是接班人，而不是变革者。”**换言之**，一个节目的“心”的意义是大于“脸”的意义的；“换脸”未必就是“换心”；但《新闻联播》目前还只能接班性地“换脸”而不能变革性地“换心”。

处理关联句子的方法有三种：

- 调整关联句的权重，使更重要的句子优先成为摘要句。
- 调整关联句的权重，使关联的两个句子都成为或都不成为摘要句。
- 输出摘要时，如果不能完整的保持相关联的句子，则删除句前的关联词。

句子间的关联通过关联性的词语来表示。处理关联句可以根据关联性词语的类型分别处理。表 5-5 列出了各种关联类型的处理方法。

表 5-5 关联词表

关 联 类 型	关 联 词	处 理 方 式
转折	虽然…但是…	对于这类偏正关系的，调整后面部分的关键句的权重，保证其大于前面部分的权重。当只有一句是摘要句时，删除该句前的关联词
因果	因为…所以…/因此	
递进	不但…而且… 尤其	
并列	一方面…另一方面…	对于这类并列关系，使关键句的权重都一样。找不到对应的关联句的删除该句子前面的关联词
承接	接着 然后	
选择	或者…或者	
分述	首先…其次…	
总述	总而言之 综上所述 总之	这类可承前省略的，如果与前面的句子都是摘要句，则保持不变。否则，如果前面的句子不是摘要句则删除该句子前面的关联词
等价	也就是说 即 换言之	
话题转移	另外	
对比	相对而言	
举例说明	比如 例如	这类可承前省略的，如果与前面的句子都是摘要句，则保持不变。否则删除后面的句子

定义句子的关系类型：

```
public enum RelationType {
    conjunctive, //偏正
    juxtapose, //并列
    conclusion //承前省略
}
```

表示句子及它们之间的关系：

```
public class SentenceRelation {
    SentenceScore pre; //前一个句子
    SentenceScore sub; //下一个句子
    RelationType type; //关系类型

    public SentenceRelation(SentenceScore preSentence,
                           SentenceScore subSentence,
                           RelationType t) {
        pre = preSentence;
        sub = subSentence;
        type = t;
    }
}
```



### 5.9.3 Lucene 中的动态摘要

Lucene 扩展包中有一个实现自动摘要的包——HighLighter。HighLighter 返回一个或多个和搜索关键词最相关的段落。其实现原理是：首先由分段器（Fragmenter）把文本分成多个段落，然后 QueryScorer 计算每个段落的分值。QueryScorer 只包含需要做高亮显示的 Term。

为了实现高亮显示，以 lucene-3.0.2 为例，除了依赖 lucene-core-3.0.2.jar 和 lucene-highlighter-3.0.2.jar 以外，还依赖 lucene-memory-3.0.2.jar。通过调用 getBestFragments 方法返回一个或多个和搜索关键词最相关的段落。实现高亮显示最简单的做法如下所示：

```
TokenStream tokenStream = analyzer.tokenStream("title", new StringReader(
    title));
String highLightText = highlighter.getBestFragment(tokenStream, title);
```

搜索时使用 analyzer 分析出搜索词会导致搜索速度变慢。形成索引的时候已经分过词了，因此可以在索引时存储位置信息。可以通过 IndexReader 取出索引中保存的词的位置信息，通过词的位置信息来构造 TokenStream，这样就避免了搜索时再次分词导致的搜索速度降低。一般情况下，用户经常使用关键词搜索，而索引只需要做一次就可以了，所以提升搜索速度很重要。如果 IndexReader 中的 Token 位置有重叠，为了把冗余的 Token 去掉，TokenStream 构造起来会麻烦一些。

对标题列高亮显示的实现代码如下所示：

```
SimpleHTMLFormatter simpleHTMLFormatter =
    new SimpleHTMLFormatter("<font color='red'>", "</font>");
Highlighter highlighter = new Highlighter(simpleHTMLFormatter, new
    QueryScorer(query));
highlighter.setTextFragmenter(new SimpleFragmenter(40));
for (int i = 0; i < hits.length; i++){
    Document hitDoc = isearcher.doc(hits[i].doc);
    String text = hitDoc.get("title");
    TermPositionVector tpv =
        (TermPositionVector)isearcher.getIndexReader().getTermFreqVector
        (hits[i].doc, "title");
    TokenStream tokenStream=TokenSources.getTokenStream(tpv);
    String highLightText =highlighter.getBestFragment(tokenStream,
        text);
    System.out.println(highLightText);
}
```

FastVectorHighlighter 是一个快速的高亮工具，相对于 Highlighter 它有三个好处：

- FastVectorHighlighter 可以支持  $n$  元分词器分出来的列；
- FastVectorHighlighter 可以输出不同颜色的高亮效果；
- FastVectorHighlighter 可以对词组高亮。如检索 lazy dog，FastVectorHighlighter 返回结果是<b>lazy dog</b>，而 Highlighter 返回的结果则是<b>dog</b>。

但 FastVectorHighlighter 不支持所有的查询，例如 WildcardQuery 或 SpanQuery 等。

可以使用内存索引来测试 FastVectorHighlighter 高亮显示的效果。

```

FragListBuilder fragListBuilder = new SimpleFragListBuilder();
//创建多颜色标签 ScoreOrderFragmentsBuilder
FragmentsBuilder fragmentBuilder = new ScoreOrderFragmentsBuilder(
    BaseFragmentsBuilder.COLORED_PRE_TAGS,
    BaseFragmentsBuilder.COLORED_POST_TAGS);
FastVectorHighlighter highlighter = new FastVectorHighlighter(true, true,
    fragListBuilder, fragmentBuilder); //创建 FastVectorHighlighter 实例

FieldQuery fieldQuery = highlighter.getFieldQuery(titleQuery);
//创建 FieldQuery
String highLightText = highlighter.getBestFragment(
    fieldQuery, isearcher.getIndexReader(),
    hits[i].doc, "title", 10000); //高亮片断

```

FastVectorHighlighter 性能很好，但是 SimpleFragListBuilder 硬编码了 6 个字符的边界，导致匹配短文本时，左边的内容显示不全。修改后的 SimpleFragListBuilder 如下所示：

```

public class SimpleFragListBuilder implements FragListBuilder {
    public static final int MARGIN = 6;
    public static final int MIN_FRAG_CHAR_SIZE = MARGIN * 3;

    public FieldFragList createFieldFragList(FieldPhraseList field
        PhraseList,
        int fragCharSize) {
        if (fragCharSize < MIN_FRAG_CHAR_SIZE)
            throw new IllegalArgumentException("fragCharSize(" +
                fragCharSize
                    + ") is too small. It must be " + MIN_FRAG_CHAR_SIZE
                    + " or higher.");

        FieldFragList ffl = new FieldFragList(fragCharSize);

```





```
List<WeightedPhraseInfo> wpil = new ArrayList<WeightedPhraseInfo>();
Iterator<WeightedPhraseInfo> ite = fieldPhraseList.phraseList.
iterator();
WeightedPhraseInfo phraseInfo = null;
int startOffset = 0;
boolean taken = false;
while (true) {
    if (!taken) {
        if (!ite.hasNext())
            break;
        phraseInfo = ite.next();
    }
    taken = false;
    if (phraseInfo == null)
        break;
    //如果当前短语超过了前一个段落的边界，
    //就放弃这个短语，然后试下一个短语
    if (phraseInfo.getStartOffset() < startOffset)
        continue;
    wpil.clear();
    wpil.add(phraseInfo);
    int firstOffset = phraseInfo.getStartOffset();
    int st = phraseInfo.getStartOffset() - MARGIN <
start Offset ? startOffset
        : phraseInfo.getStartOffset() - MARGIN;
    int en = st + fragCharSize;
    if (phraseInfo.getEndOffset() > en)
        en = phraseInfo.getEndOffset();

    int lastEndOffset = phraseInfo.getEndOffset();
    while (true) {
        if (ite.hasNext()) {
            phraseInfo = ite.next();
            taken = true;
            if (phraseInfo == null)
                break;
        } else
            break;
        if (phraseInfo.getEndOffset() <= en){
            wpil.add(phraseInfo);
            lastEndOffset = phraseInfo.getEndOffset();
        } else
            break;
    }
    int matchLen = lastEndOffset - firstOffset;
    //现在重新计算开始和结束位置
```

```

        int newMargin = (fragCharSize - matchLen) / 2;
        st = firstOffset - newMargin;
        if (st < startOffset) {
            st = startOffset;
        }
        en = st + fragCharSize;
        startOffset = en;
        ffl.add(st, en, wpil);
    }
    return ffl;
}
}

```



## 5.10 文本分类

文本分类程序把一个未见过的文档分成已知类别中的一个或多个。利用文本分类技术可以对网页分类，也可以用于为用户提供个性化新闻或者垃圾邮件过滤。

另外一个应用的例子，考虑划分商品的等级，譬如车有高档车、中档车、低档车，可以看成是按价格分类的例子。

为了理解机器学习的算法如何对文本分类，首先看一下人是如何对事物分类的。为了判断食物是否为健康食品，可参考食品中的饱和脂肪、胆固醇、糖和钠的含量。例如把食品分类成“健康食品”和“不健康食品”。如果这些值超过一个阈值就认为该食品是“不健康的”，否则是“健康的”。首先找出一些重要的特征，然后从每个待分类的项目中寻找特征，从抽取出的特征中组合证据（combine evidence），最后根据组合证据按照某种决策机制对项目分类。

在食品分类的例子中，特征是饱和脂肪、胆固醇、糖和钠的含量。可以通过阅读打印在食品包装上的营养成分表来取得待分类食品的特征对应的值。

为了量化食物的健康程度（记做  $H$ ），有很多方法来组合证据，最简单的方法是按权重求和。

$$H(\text{食物}) = W_{\text{脂肪}} \text{脂肪}(\text{食物}) + W_{\text{胆固醇}} \text{胆固醇}(\text{食物}) + W_{\text{糖}} \text{糖}(\text{食物}) + W_{\text{钠}} \text{钠}(\text{食物})$$

这里  $W_{\text{脂肪}}$ 、 $W_{\text{胆固醇}}$  等是和每个特征关联的重要度。在这个公式里，这些值可能是负数。

把给定的文档归到两个类别中的一个叫做两类分类，例如垃圾邮件过滤，就只需要确定“是”还是“不是”垃圾邮件。分到多个类别中的一个叫做多类分类，例如中图法分类目录把图书分成 22 个基本大类。

首先准备好训练文本集，也就是一些已经分好类的文本。每个类别路径下包含属于该类别的一些文本文件。例如文本路径在“D:\train”，类别路径是：

```
D:\train\zippo 收藏乐器
D:\train\宠物玩具音像书
D:\train\电脑数码产品办公设备
D:\train\房屋
D:\train\服装箱包首饰钟表眼镜
D:\train\家电家具居家
D:\train\交友婚介祝福寻人寻物
D:\train\教育培训
D:\train\美容按摩医药
D:\train\汽车交通
D:\train\人才职场
D:\train\生活服务
D:\train\手机电话卡号
D:\train\玩运动出游
```

例如：“教育培训”类别路径下包含属于“教育培训”类别的一些文本文件，每个文本文件叫做一个实例（instance）。训练文本文件可以手工整理一些或者从网络定向抓取新闻网站上已经按栏目分好类的新闻。

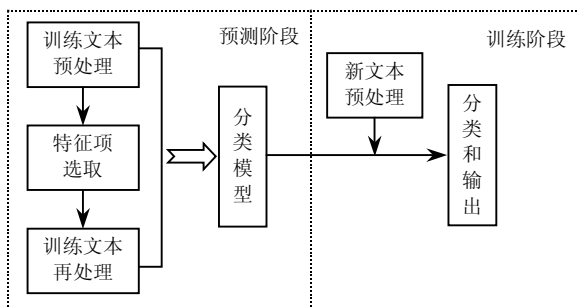


图 5-12 文本分类程序框架

文本分类主要分为训练阶段和预测阶段。一个典型的文本分类程序框架如图 5-12 所示。

常见的分类方法有支持向量机（SVM）、K 个最近的邻居（KNN）和朴素贝叶斯（Naive Bayes）等，可以根据应用场景选择合适的文本分类方法。例如，支持向量机适合对长文本分类；朴素贝叶斯对短文本分类准确度较高。

为了加快分类的执行速度，可以在训练阶段输出分类模型文件，这样在预测新文本类别阶段就不再需要直接访问训练文本集，只需要读取已经保存在分类模型文件中的信息即

可。可以在预测之前，把分类模型文件预加载到内存中。

交叉验证（Cross Validation）是用来验证分类器性能的一种统计分析方法。其基本思想是把在某种意义下将原始数据集进行分组，一部分作为训练集，另一部分作为验证集，首先用训练集对分类器进行训练，再利用验证集来测试训练得到的模型，以此来作为评价分类器的性能指标。

### 5.10.1 特征提取

待分类的文本往往包括很多单词，这些单词对分类没有太大的贡献，所以需要提取特征词。可以按词性过滤，只选择某些词性作为分类特征，比如说，只选择名词和动词作为分类特征词。文本分类的精度随分类特征词的个数持续提高，一般至少可以选 2000 个分类特征词。

分类特征并不一定就是一个词。例如，可以通过检查标题和签名把文本分类成是否是信件内容。可以看标题是否包含“来自\*\*”和“致\*\*”地址，内容结束处是否包含日期和问候用语等。这样的特征集合仅适用于信件类别。

特征选择的常用方法还有 CHI 方法和信息增益（Information Gain）方法等。首先介绍特征选择的 CHI 方法。

利用 CHI 方法来进行特征抽取是基于如下假设：在指定类别文本中出现频率高的词条与在其他类别文本中出现频率比较高的词条，对判定文档是否属于该类别都是很有帮助的。

CHI 方法衡量单词 *term* 和类别 *class* 之间的依赖关系。如果 *term* 和 *class* 是互相独立的，则该值接近于 0。一个单词的 CHI 统计通过表 5-6 计算。

表 5-6 CHI 统计变量定义表

	属于 <i>class</i> 类	不属于 <i>class</i> 类	合 计
包含单词 <i>term</i>	<i>a</i>	<i>b</i>	<i>a+b</i>
不含单词 <i>term</i>	<i>c</i>	<i>d</i>	<i>c+d</i>
合计	<i>a+c</i>	<i>b+d</i>	<i>a+b+c+d=n</i>

其中，*a* 表示属于类别 *class* 的文档集合中出现单词 *term* 的文档数；*b* 表示不属于类别 *class* 的文档集合中出现单词 *term* 的文档数；*c* 表示属于类别 *class* 的文档集合中没有出现单词 *term* 的文档数；*d* 表示不属于类别 *class* 的文档集合中没有出现单词 *term* 的文档数；*n* 代表文档总数。



表 5-6 中的一个单词 *term* 的 CHI 统计公式如下所示：

$$\text{chi\_statistics}(\text{term}, \text{class}) = n * (ad - cb)^2 / ((a + c) * (b + d) * (a + b) * (c + d))$$

类别 *class* 越依赖单词 *term*，则 CHI 统计值越大。

表 5-6 也叫做相依表（contingency table）。开源自然语言处理项目 *minorthird* 中 *ContingencyTable* 类的 CHI 统计实现代码如下所示：

```
//取 log 避免溢出
public double getChiSquared(){
    double n = Math.log(total());
    double num = 2*Math.log(Math.abs((a*d) - (b*c)));
    double den = Math.log(a+b)+Math.log(a+c)+Math.log(c+d)+Math.
        log(b+d);
    double tmp = n+num-den;
    return Math.exp(tmp);
}
```

计算每个特征对应的 CHI 值的实现代码如下所示：

```
for (Iterator<Feature> i=index.featureIterator(); i.hasNext(); )
//遍历特征集合
{
    Feature f = i.next();
    int a = index.size(f,ExampleSchema.POS_CLASS_NAME);
    //正类中包含特征的文档数
    int b = index.size(f,ExampleSchema.NEG_CLASS_NAME);
    //负类中包含特征的文档数
    int c = totalPos - a; //正类中不包含特征的文档数
    int d = totalNeg - b; //负类中不包含特征的文档数

    ContingencyTable ct = new ContingencyTable(a,b,c,d);
    double chiScore = ct.getChiSquared();//计算特征的 CHI 值
    filter.addFeature( chiScore,f );
}
```

如果这里的 *a+c* 或 *b+d* 或 *a+b* 或 *c+d* 中的任意一个值为 0，则会导致除以零溢出的错误。由于数据稀疏导致了这样的问题，可以采用平滑算法来解决。

对所有的候选特征词，按上面得到的特征区分度排序，如果候选特征词的个数大于 5000，则选取前 5000，否则选取所有特征词。

ChiSquareTransformLearner 测试特征提取:

```
Dataset dataset = CnSampleDatasets.sampleData("toy", false);
System.out.println( "old data:\n" + dataset );
ChiSquareTransformLearner learner = new ChiSquareTransformLearner();
ChiSquareInstanceTransform filter =
    (ChiSquareInstanceTransform) learner.batchTrain( dataset );
filter.setNumberOfFeatures(10);
dataset = filter.transform( dataset );
System.out.println( "new data:\n" + dataset );
```

信息增益 (Information Gain) 是广泛使用的特征选择方法。在信息论中, 信息增益的概念是: 某个特征的值对分类结果的确定程度增加了多少。

信息增益的计算方法是: 把文档集合  $D$  看成一个符合某种概率分布的信息源, 依靠文档集合的信息熵和文档中词语的条件熵之间信息量的增益关系确定该词语在文本分类中能提供的信息量。

词语  $w$  的信息量的计算公式为:

$$IG(w)=H(D)-H(D|w)=$$

$$-\sum_{d_i \in D} P(d_i) \times \log_2 P(d_i) + \sum_{w \in \{0,1\}} P(w) \sum_{d_i \in D} P(d_i | w) \times \log_2 P(d_i | w)$$

计算一个属性的熵:

```
//输入参数 p 是属性的出现次数分布, tot 是属性出现的总次数
public double Entropy(double[] p, double tot){
    double entropy = 0.0;
    for (int i=0; i<p.length; i++) {
        if (p[i]>0.0) { entropy += -p[i]/tot *Math.log(p[i]/tot) /Math.log
            (2.0); }
    }
    return entropy;
}
```

计算所有特征的熵:

```
double[] classCnt = new double[ N ];
double totalCnt = 0.0;
for (int c=0; c<N; c++)//循环遍历所有的类别
```



```
{
    classCnt[c] = (double)index.size(schema.getClassName(c)); //每类的文档数
    totalCnt += classCnt[c]; //总文档数
}
double totalEntropy = Entropy(classCnt,totalCnt); //训练文档的总熵值

for (Iterator<Feature> i=index.featureIterator(); i.hasNext(); ) {
    Feature f = i.next();
    double[] featureCntWithF = new double[ N ]; //出现特征的文档在不同
    //类别中的分布
    double[] featureCntWithoutF = new double[ N ]; //不出现特征的文档在不同
    //类别中的分布
    double totalCntWithF = 0.0;
    double totalCntWithoutF = 0.0;

    for (int c=0; c<N; c++) {
        featureCntWithF[c] = (double)index.size(f,schema.getClassName(c));
        featureCntWithoutF[c] = classCnt[c] - featureCntWithF[c];
        totalCntWithF += featureCntWithF[c];
        totalCntWithoutF += featureCntWithoutF[c];
    }

    double entropyWithF = Entropy(featureCntWithF,totalCntWithF);
    //出现特征的熵
    //不出现特征的熵
    double entropyWithoutF = Entropy(featureCntWithoutF,totalCntWithoutF);

    double wf = totalCntWithF /totalCnt; //出现词的概率
    //特征的信息增益
    double infoGain = totalEntropy -wf*entropyWithF -(1.0-wf)*entropy
    WithoutF;
    igValues.add( new IGPair(infoGain,f) );
}
```

### 5.10.2 中心向量法

中心向量法又叫 Rocchio 方法。中心向量法使用标准的 TF/IDF 带权重的向量表示文档。通过把训练集文档中的向量加在一起对每一个类别计算一个原型向量。把测试文档和每一个类别的原型向量计算相似度，把该文档归为相似度最近的原型向量所代表的类。文档在向量空间中的表示如图 5-13 所示。

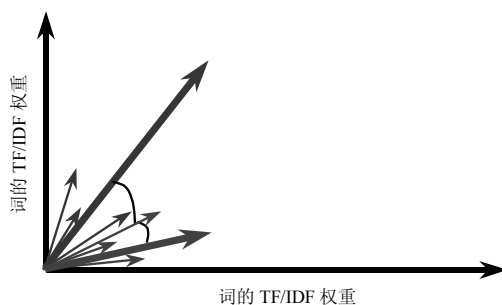


图 5-13 向量空间中的文档相似度

原型向量和测试文档的相似度衡量可以用夹角余弦或距离的方法来计算。原型向量  $V(x)$  和文档向量  $V(y)$  的夹角余弦相似度计算如下：

$$\text{cosine-similarity}(x, y) = \frac{V(x) \cdot V(y)}{|V(x)| |V(y)|}$$

这里：

- 原型向量  $V(x) = \langle w(t_1, x), w(t_2, x), \dots, w(t_n, x) \rangle$ ;
- 文档向量  $V(y) = \langle w(t_1, y), w(t_2, y), \dots, w(t_n, y) \rangle$ ;
- $V(x) \cdot V(y)$  是两个带权重的向量的点乘积，计算方法是  $V(x) \cdot V(y) = w(t_1, x) * w(t_1, y) + w(t_2, x) * w(t_2, y) + \dots + w(t_n, x) * w(t_n, y)$ ;
- $|V(x)|$  和  $|V(y)|$  表示欧几里得范数。例如  $|V(x)| = \sqrt{t_1^2 + t_2^2 + \dots + t_n^2}$ ，这里的  $t$  是文档  $x$  中出现的词的权重。

开源项目 Classifier4J (<http://classifier4j.sourceforge.net/>) 中包含中心向量法的文本分类算法实现代码如下所示：

```
//计算两个向量的点乘积
public static int scalarProduct(int[] one, int[] two)
    throws IllegalArgumentException {
    if (one.length != two.length) {
        throw new IllegalArgumentException("两个向量长度不一样");
    }

    int result = 0;
    for (int i = 0; i < one.length; i++) {
        result += one[i] * two[i];
    }
}
```





```
        return result;
    }

    //计算向量长度
    public static double vectorLength(int[] vector)
        throws IllegalArgumentException {
        double sumOfSquares = 0d;
        for (int i = 0; i < vector.length; i++) {
            sumOfSquares = sumOfSquares + (vector[i] * vector[i]);
        }

        return Math.sqrt(sumOfSquares);
    }

    //计算夹角余弦值
    public static double cosineOfVectors(int[] one, int[] two) {
        if (one.length != two.length) {
            throw new IllegalArgumentException("两个向量长度不一样");
        }
        double denominator = (vectorLength(one) * vectorLength(two));
        if (denominator == 0) {
            return 0;
        } else {
            return (scalarProduct(one, two)/denominator);
        }
    }
}
```

关于稀疏数据的问题，比如 SVM，其数据存储格式为 `index:val`，只包含非零的 `val`。但是如果放到内存中运算怎么办，比如两个文章的 VSM 模型，要计算余弦相似度，难道需要把 0 位再补上么？

为了快速计算两个数组  $x$  和  $y$  的点积，只计算相同位置不为 0 的值就可以了。对于稀疏维度，为了节省空间 `<Index,Value>` 不采用哈希表存储，而是将 `index` 全部按升序排列，然后通过归并两个排好序的数组来计算。有点类似搜索引擎中的 `docList` 集合求 AND 的操作。计算的时间复杂度是  $O(m+n)$ 。计算点积的实现代码如下所示：

```
static double scalarProduct(Node[] x, Node[] y){
    double sum = 0;
    int xlen = x.length;
    int ylen = y.length;
    int i = 0;
    int j = 0;
    while(i < xlen && j < ylen){
```

```

        if(x[i].index == y[j].index)
            sum += x[i++].value * y[j++].value;
        else
        {
            if(x[i].index > y[j].index)
                ++j;
            else
                ++i;
        }
    }
    return sum;
}

```

分类测试代码如下所示：

```

//定义存储向量的变量
TermVectorStorage storage = new HashMapTermVectorStorage();
//新建一个向量分类器
VectorClassifier vc = new VectorClassifier(storage);
//定义一个叫做“test”的类别
String category = "test";
//训练一个句子属于类别“test”
vc.teachMatch(category, sentence1);
//doc1 和类别 category 的 cos 夹角距离大约是 0.852。
assertEquals(0.852d, vc.classify(category, doc1), 0.001);
//doc2 和类别 category 的 cos 夹角距离大约是 0.301。
assertEquals(0.301d, vc.classify(category, doc2), 0.001);
//“bye”和类别 category 的 cos 夹角距离是 0。
assertEquals(0.0d, vc.classify(category, "bye"), 0.001);
//“bye”和类别“does not exist”的 cos 夹角距离是 0。
assertEquals(0.0d, vc.classify("does not exist", "bye"), 0.001);

```

中心向量法的优点是容易实现且计算简单，但是对于类别文档比较分散的情况效果不好，实用的分类系统很少采用这种算法。

### 5.10.3 朴素贝叶斯

考虑文档  $d$  属于类别  $c$  的概率。一个文档  $d$  的概率是  $P(d)$ ，文档  $d$  正好属于类别  $d$  的条件概率是  $P(c|d)$ 。一个文档  $d$  属于类别  $c$  的联合概率  $P(c,d)$  有如下两种计算方式：

$$P(c,d) = P(c|d) * P(d) = P(d|c) * P(c)$$

根据上面的等式得到贝叶斯理论的一般形式：



$$P(C|D) = \frac{P(D|C)P(C)}{P(D)} = \frac{P(D|C)P(C)}{\sum_{c \in C} P(D|C=c)P(C=c)}$$

其中  $C$  和  $D$  是随机变量。

对文本分类就是在所有的类别中，找到最大的条件概率  $P(c|d)$  对应的类别  $c$ 。形式化的写法是： $\arg \max P(c|d)$ 。根据贝叶斯理论可以得到：

$$Class(d) = \arg \max_{c \in C} P(c|d) = \arg \max_{c \in C} \frac{P(d|c)P(c)}{\sum_{c \in C} P(d|c)P(c)}$$

这就是贝叶斯分类公式。其含义是：在所有可能的类集合  $C$  中返回类  $c$ ，使得  $P(c|d)$  最大。这里并不直接计算  $P(c|d)$ ，根据贝叶斯理论，可以通过计算  $P(d|c)$  和  $P(c)$  得到  $P(c|d)$ 。

为了简化计算过程，朴素贝叶斯模型假定特征变量是相互独立的，也就是待分类文本中的词与词之间没有关联。假设  $d=w_1, w_2, \dots, w_n$ ，则  $P(d|c) = \prod_{i=1}^n P(w_i|c)$ 。

给定一个类  $c$ ，我们为每个词定义一个布尔型的随机变量  $w_i$ 。布尔型事件的结果是 0 或 1。 $P(w_i=1|c)$  的概率可以说是项  $w_i$  通过类  $c$  产生的概率。相反地  $P(w_i=0|c)$  的概率可以说是项  $w_i$  不通过类  $c$  产生的概率。这就是多变量伯努利（Multiple Bernoulli）事件空间。

在这个事件空间下，为每个项在某些类  $c$  下，估计这个词是由这个类生成的可能性。例如，在垃圾分类中， $P(\text{cheap}=1|\text{spam})$  可能有很高的概率，但是  $P(\text{dinner}=1|\text{spam})$  将有一个很低的概率。

图 5-14 显示了怎样设置训练文档能被呈现在这个事件空间中。在这个例子中有 10 个文档，每个文档用唯一的 id 标识、两类（spam 和 not spam），和包含“cheap”、“buy”、“banking”、“dinner”和“the”项的词汇表构成。在这个例子中  $P(\text{spam})=3/10$ ， $P(\text{not spam})=7/10$ 。接着，必须估计每个词和类搭配的  $P(w|c)$ 。最直接的方法是使用最大似然法估计概率，公式如下：

$$P(w|c) = \frac{df_{w,c}}{N_c}$$

文档 id	cheap	buy	banking	dinner	the	class
1	0	0	0	0	1	not spam
2	1	0	1	0	1	spam
3	0	0	0	0	1	not spam
4	1	0	1	0	1	spam
5	1	1	0	0	1	spam
6	0	0	1	0	1	not spam
7	0	1	1	0	1	not spam
8	0	1	0	0	1	not spam
9	0	0	0	0	1	not spam
10	1	0	0	1	1	not spam

图 5-14 在多变量伯努利事件空间中表示文档

这里,  $df_{w,c}$  是在类  $c$  中包含词  $w$  的训练文档的数量,  $N_c$  是类  $c$  的训练文档的总数。最大似然估计无非是在类别  $c$  中包含项  $w$  的文档的比例。使用最大似然估计, 容易计算  $P(\text{the}|\text{spam})=1$ ,  $P(\text{the}|\text{not spam})=1$ ,  $P(\text{dinner}|\text{spam})=0$ ,  $P(\text{dinner}|\text{not spam})=1/7$  等。

使用多变量伯努利模型, 文档似然值  $P(d|c)$  可以写为:

$$P(d|c) = \prod_{w \in V} P(w|c)^{\delta(w,d)} (1 - P(w|c))^{1-\delta(w,d)}$$

其中当且仅当在文档  $d$  中出现词  $w$  时,  $\delta(w,d)$  是 1。

实际上, 由于 0 概率问题, 所以不可能使用最大似然估计。为了解释 0 概率问题, 让我们回到图 5-14 中的垃圾分类的例子。假设我们接收一封垃圾邮件包含“dinner”一词。无论电子邮件包含或不包含其他的词,  $P(d|c)$  一直是 0, 因为  $P(\text{dinner}|\text{spam})=0$ , 而这个词在文档中出现 (也就是  $\delta(\text{dinner}, d)=1$ ), 因此任何包含词“dinner”的文档将都会自动计算成垃圾的概率是零。这个问题比较普遍, 因为每当一个文档包含一个词, 而那个词从来不出现在一个或多个类时, 零概率问题在于会产生。这里的问题是最大似然估计是基于训练集中的出现计数。然而, 这个训练集是有限的, 因此不可能观察到所有可能的事件, 这就是所谓的数据稀疏。稀疏往往是因为训练集太小, 但是也会发生在比较大的数据集上。因此, 我们必须改变这样一种方式的估计, 对所有词, 包括那些没有在给定类中观察到的, 给予一些概率量。我们必须为所有在词典中的项确保  $P(w|c)$  是非零的。通过这样做, 就能避免所有与零概率相关的问题。平滑技术可以克服零概率问题, 一个流行的平滑技术是贝叶斯平滑。贝叶斯平滑假设模型上的某些先验概率并使用最大后验估计 (max a posterior)。由此产生平滑估计的多变量伯努利模型的形式:

$$P(w|c) = \frac{df_{w,c} + \alpha_w}{N_c + \alpha_w + \beta_w}$$

$\alpha_w$  和  $\beta_w$  是依赖于  $w$  的参数。不同的参数设置导致不同的估计结果。一种流行的选择是对所有  $w$  设置  $\alpha_w = 1$  和  $\beta_w = 0$ 。结果是以下的估计：

$$P(w|c) = \frac{df_{w,c} + 1}{N_c + 1}$$

另一个选择是，对所有的  $w$  设置  $\alpha_w = \mu \frac{N_w}{N}$  和  $\beta_w = \mu(1 - \frac{N_w}{N})$ 。其中  $N_w$  是在  $w$  出现时训练文档的总数， $\mu$  是可调参数。结果是如下估计：

$$P(w|c) = \frac{df_{w,c} + \mu \frac{N_w}{N}}{N_c + \mu}$$

此事件空间只记录一个词是否出现；它没有记录这个词出现了多少次，但词频是一个重要的信息，对长文本来说尤其如此。现在我们描述一个考虑到频率的多项（multinomial）事件空间。

在图 5-15 的例子中有 10 个文档（每个文档用唯一的 id 标识）、两类（spam 和 not spam），和包含“cheap”、“buy”、“banking”、“dinner”和“the”这些项的词汇表。和多变量伯努利表示唯一的区别是事件不再是布尔型的。多项模型的最大似然估计和多变量伯努利模型很相似，公式是：

$$P(w|c) = \frac{tf_{w,c}}{|c|}$$

文档 id	cheap	buy	banking	dinner	the	class
1	0	0	0	0	2	not spam
2	3	0	1	0	1	spam
3	0	0	0	0	1	not spam
4	2	0	3	0	2	spam
5	5	2	0	0	1	spam
6	0	0	1	0	1	not spam
7	0	1	1	0	1	not spam
8	0	1	0	0	1	not spam
9	0	0	0	0	1	not spam
10	1	0	0	1	1	not spam

图 5-15 在多项事件空间中如何表示文档

这里  $tf_{w,c}$  是训练集中的词  $w$  在类  $c$  中出现的次数，而  $|c|$  是属于类  $c$  的词总次数。在垃圾分类例子中， $P(\text{the}|\text{spam})=4/20$ ， $P(\text{the}|\text{not spam})=9/15$ ， $P(\text{dinner}|\text{spam})=0$ ，而  $P(\text{dinner}|\text{not spam})=1/15$ 。

因为词是多项分布，给定类  $c$  的文档  $d$  的似然公式是：

$$P(d|c) = P(|d|)(tf_{w_1,d}, tf_{w_2,d}, \dots, tf_{w_v,d})! \prod_{w \in V} P(w|c)^{tf_{w,d}}$$

这里  $tf_{w,d}$  是词  $w$  在文档  $d$  中出现的次数，而  $|d|$  是出现在文档  $d$  中的词总次数。 $P(|d|)$  是产生长度是  $|d|$  的文档的概率，而  $(tf_{w_1,d}, tf_{w_2,d}, \dots, tf_{w_v,d})!$  是多项系数。注意到  $P(|d|)$  和多项系数是依赖于文档的，为了分类的目的，可以省略掉这两项。因此实际需要计算的是  $\prod_{w \in V} P(w|c)^{tf_{w,d}}$ 。

词似然值的贝叶斯平滑估计根据如下公式计算：

$$P(w|c) = \frac{tf_{w,c} + \alpha_w}{|c| + \sum_{w \in V} \alpha_w}$$

这里  $\alpha_w$  是一个依赖于  $w$  的参数。对所有的  $w$ ，设置  $\alpha_w=1$  是一种可能的选择。对应如下的估计：

$$P(w|c) = \frac{tf_{w,c} + 1}{|c| + |V|}$$

另一个流行的选择是设置  $\alpha_w = \mu \frac{cf_w}{|c|}$ ，这里  $cf_w$  是词  $w$  出现在训练文档中的总次数。 $|c|$  是词在所有的训练文档中出现的总次数，而  $\mu$  则是一个可调的参数。在这个设置下，得到如下的估计：

$$P(w|c) = \frac{tf_{w,c} + \mu \frac{cf_w}{|c|}}{|c| + \mu}$$

多变量伯努利（Multiple Bernoulli）模型又叫做文档型模型。多项（multinomial）模型又叫做词频型模型。这里实现文档型分类模型。



计算先验概率（Prior Probability）。

```
private static TrainingData trainingData=TrainingData.getInstance();
//得到训练集

/**
 * 先验概率
 * @param c 给定的分类
 * @return 给定条件下的先验概率
 */
public static float calculatePc(String c) {
    float Nc = trainingData.getClassDocNum(c); //给定分类的训练文本数
    float N = trainingData.getTotalNum(); //训练集中文本总数
    return (Nc / N);
}
```

然后计算分类条件概率（Conditional Probability）。

类条件概率  $P(w_j | c_j) = (N(W = w_i, C = c_j) + 1) / (N(C = c_j) + M + V)$

其中， $N(W=w_i, C=c_j)$ 表示类别  $c_j$ 中包含词  $w_i$ 的训练文本数量； $N(C=c_j)$ 表示类别  $c_j$ 中的训练文本数量； $M$ 值用于平滑，避免由于  $N(W=w_i, C=c_j)$ 过小所引发的问题； $V$ 表示类别的总数。

```
/**
 * 计算类条件概率
 * @param w 给定的词
 * @param c 给定的分类
 * @return 给定条件下的类条件概率
 */
public static float calculatePwc(String w, String c) {
    //返回给定分类中包含分类特征词的训练文本的数目
    float dfwc = tdm.getCountContainKeyOfClassification(c, w);
    //返回训练文本集中在给定分类下的训练文本数目
    float Nc = tdm.getClassDocNum(c);
    //类别数量
    float V = tdm.getTraningClassifications().length;
    return ( (dfwc + 1) / (Nc + M + V));
}
```

利用样本数据集计算先验概率和各个文本向量属性在分类中的条件概率，从而计算出各个概率值，最后对各个概率值进行排序，选出最大的概率值，即为所属的分类。

为了避免结果过小，对乘积的结果取对数，因此计算整个概率的公式是： $\log(P(d|c)) + \log(c)$ 。

```
/**
 * @param d 给定文本的属性向量
 * @param Cj 给定的类别
 * @return 类别概率
 */
float calcProd(String[] d, String Cj) {
    float ret = 0.0F;
    //类条件概率连乘
    for (int i = 0; i < d.length; i++) {
        {
            String wi = d[i];
            //因为连乘结果过小，所以把转成取对数
            //对分类的最终结果无影响，因为只是比较概率大小而已
            ret += Math.log(ClassConditionalProbability.calculatePwc(wi, Cj));
        }
    }
    //再乘以先验概率
    ret += Math.log(PriorProbability.calculatePc(Cj));
    return ret;
}
```

最后取最大的概率对应的类：

```
String[] classes = tdm.getTraningClassifications();//返回所有的类名
float probability = 0.0F;
List<ClassifyResult> crs = new ArrayList<ClassifyResult>();//分类结果
for (int i = 0; i < classes.length; i++){
    {
        String ci = classes[i];//第 i 个分类
        //计算给定的文本属性向量 terms 在给定的分类 Ci 中的分类条件概率
        probability = calcProd(terms, ci);
        //保存分类结果
        ClassifyResult cr = new ClassifyResult(ci, probability);
        crs.add(cr);
    }
}
//返回概率最大的分类
float maxPro = crs.get(0).probability;
String c = crs.get(0).classification;
for(ClassifyResult cr:crs) {
    {
        if(cr.probability > maxPro) {
            {
```



```

        c = cr.classification;
        maxPro = cr.probability;
    }
}
return c;

```

#### 5.10.4 支持向量机

和贝叶斯分类器基于概率来分类不同，支持向量机（SVM）是基于几何学原理来实现分类的。假设“+”是特征空间的一类点，而“-”是特征空间中的另外一类点。SVM 通过分类面来判断特征空间中待分类点的类别。其基本思想可用图 5-16 的两维情况说明。

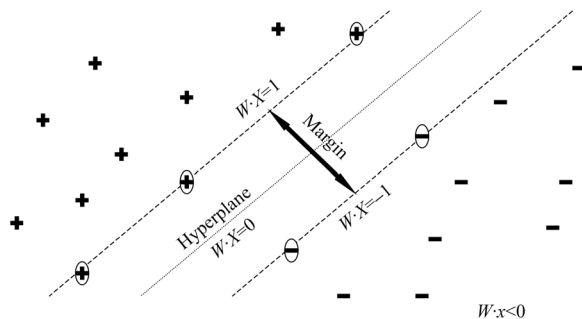


图 5-16 分类间隔

假设有  $N$  个点在  $p$  维特征空间  $X=\{x_1, x_2, \dots, x_p\}$  中分别属于两类： $C_+$  和  $C_-$ 。要解决的问题是找到一个函数  $f(x_1, x_2, \dots, x_p)$  判别出两类，对于  $C_+$  类的点返回正值，而对于  $C_-$  类的点返回负值。这个函数叫做判别函数（discriminant function）。

如果判别函数是线性的，则可以把判别函数看成如下形式：

$$f(x_1, x_2, \dots, x_p) = w_1 * x_1 + w_2 * x_2 + \dots + w_p * x_p + b$$

假设向量  $w=(w_1, w_2, \dots, w_p)$ ，向量  $x=(x_1, x_2, \dots, x_p)$ 。

用  $\langle w, x \rangle$  表示向量  $w$  和  $x$  之间的内积（inner product），或者叫做点乘积。因此这个函数的向量化表示  $g(x)$  的形式是：

$$g(x) = \text{sign}(\langle w, x \rangle + b)$$

$\langle w, x \rangle$  也记做  $w^T \cdot x$ ，也就是向量  $w$  的转置点乘向量  $x$ 。这里  $\text{sign}$  是符号函数。这里符号函数  $\text{sign}(a)$  的定义是当  $a > 0$ ，则返回 1；当  $a=0$ ，则返回 0；当  $a < 0$ ，则返回 -1。有时候也把符号函数写作  $\sigma$ 。

一般把  $w$  称作权重向量（weight vector），把  $b$  叫做偏移量（bias）。 $\langle w, x \rangle + b = 0$  所定义的面叫做超平面（hyperplane）。

每一个训练样本由一个向量（特征空间中的值组成的向量）和一个标记（标示出这个样本属于哪个类别）组成，记作： $D_i=(x_i, y_i)$ 。其中， $y$  的取值只有两种可能：1 和-1（分别用来表示属于  $C_+$ 类还是属于  $C_-$ 类）。

一般来说，如果超平面远离分类训练集中的点应该会最小化对新数据错误分类的风险。点  $i$  到平面  $\Pi_{w,b}$  的距离  $d(\Pi_{w,b}, x_i) = |w \cdot x_i + b| / \|w\|$ 。  $d(\Pi_{w,b}, x_i)$  有时候也写作  $\delta_i$ 。

选取  $w$  和  $b$ ，使得超平面到最近点的距离最大，也就是求解  $\max_{(w,b)} (\min_i d(\Pi_{w,b}, x_i))$ ，这里的  $\|w\|$  叫做向量  $w$  的欧几里得范数（Euclidean norm），计算公式是：

$$\|w\| = \sqrt{w_1^2 + w_2^2 + \dots + w_p^2}$$

对于距离超平面最近的  $C_+$ 类的点  $x_+$ 来说  $w^T \cdot x_+ + b = 1$ ；对于距离超平面最近的  $C_-$ 类的点  $x_-$ 来说  $w^T \cdot x_- + b = -1$ 。因此：

$$\max(w,b)(\min_i (\Pi_{w,b}, X_i)) = \frac{|w \cdot x_- + b| + |w \cdot x_+ + b|}{\|w\|}$$

也就是求解下面这个基本的问题。

在  $y_i(w^T x_i + b) \geq 1$  的条件下，求最小值： $\min_{w,b} \frac{1}{2} \|w\|^2$ 。

满足相等约束条件的这些点叫做支持向量（support vector），因为这些点在支持（约束）超平面。用拉格朗日乘子（Lagrange multiplier）来解决线性约束下的优化问题。

举个用拉格朗日乘子求解极值的例子。把一个拉格朗日乘子的函数整合进需要求最大或最小值的表达式。

例如： $f(x,y) = x+2y$  有约束条件  $g(x,y) = x^2+y^2-4 = 0$ ，则引入一个拉格朗日乘子后，对应拉格朗日函数  $L(x,y, \lambda) = x+2y + \lambda(x^2+y^2-4)$ 。求导数：

$$\frac{\partial L}{\partial x} = 1 + 2\lambda x = 0 \quad (1)$$

$$\frac{\partial L}{\partial y} = 2 + 2\lambda y = 0 \quad (2)$$

$$\frac{\partial L}{\partial \lambda} = x^2 + y^2 - 4 = 0 \quad (3)$$



首先根据 (1)、(2)、(3) 解出拉格朗日乘子  $\lambda$  的值，然后就可以计算出  $f(x,y)$  的最小值。对于线性可分的问题，可以通过二次规划 (Quadratic programming) 计算出拉格朗日乘子  $\lambda$  的值。

文本分类抽象成对空间中的点分类。下面举一个对图 5-17 二维空间中的样本点分类的简单例子。

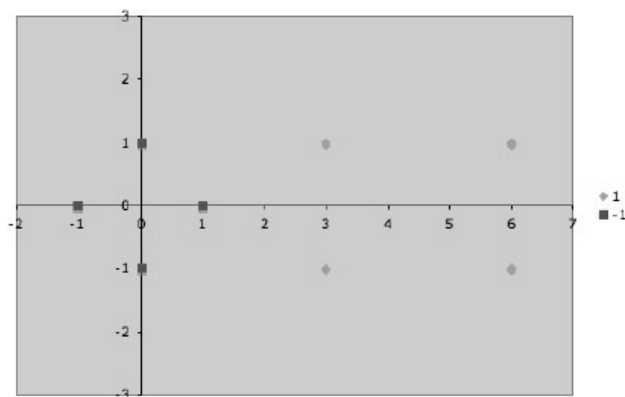


图 5-17 二维空间中的样本点

假设标志成 + 的点如下： $\{(3,1),(3,-1),(6,1),(6,-1)\}$ 。标志成 - 的点如下： $\{(1,0),(0,1),(0,-1),(-1,0)\}$ 。

因为数据是线性可分的，可以使用一个线性 SVM，也就是说  $\phi$  就是原函数。有三个支持向量： $s_1=(1,0)$ ； $s_2=(3,1)$ ； $s_3=(3,-1)$ ，增加一个偏移量输入 1，则  $s_1=(1,0,1)$ ； $s_2=(3,1,1)$ ； $s_3=(3,-1,1)$ 。

计算拉格朗日乘子  $\alpha$ 。

$$\alpha_1 * s_1 g_1 + \alpha_2 * s_2 g_1 + \alpha_3 * s_3 g_1 = -1$$

$$\alpha_1 * s_1 g_2 + \alpha_2 * s_2 g_2 + \alpha_3 * s_3 g_2 = +1$$

$$\alpha_1 * s_1 g_3 + \alpha_2 * s_2 g_3 + \alpha_3 * s_3 g_3 = +1$$

计算支持向量的点乘积，得到结果：

$$2 \alpha_1 + 4 \alpha_2 + 4 \alpha_3 = -1$$

$$4 \alpha_1 + 11 \alpha_2 + 9 \alpha_3 = +1$$

$$4 \alpha_1 + 9 \alpha_2 + 11 \alpha_3 = +1$$

求解得到:

$$\alpha_1 = -3.5, \quad \alpha_2 = 0.75, \quad \alpha_3 = 0.75$$

$w$  可以表示成支持向量的线性组合:

$$\begin{aligned} w &= \sum_i \alpha_i s_i = \alpha_1 * s_1 + \alpha_2 * s_2 + \alpha_3 * s_3 \\ &= -3.5 \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + 0.75 \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix} + 0.75 \begin{pmatrix} 3 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix} \end{aligned}$$

求解结果是  $w=(1,0)$ ,  $b=-2$ 。

设  $x_+$  是任意一个属于  $C_+$  的支持向量,  $x_-$  是任意一个属于  $C_-$  的支持向量。而且, 总是至少会存在一个  $x_+$ , 也总是至少会存在一个  $x_-$ 。

根据等式  $w^T x_+ + b = 1$  和  $w^T x_- + b = -1$  推导出  $b$  的另一个计算公式是:

$$b = -\frac{1}{2}(w^T X_+ + w^T X_-)$$

$$b = -\frac{1}{2}(w^T S_2 + w^T S_1) = -\frac{1}{2}\left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}(3,1) + \begin{pmatrix} 1 \\ 0 \end{pmatrix}(1,0)\right) = -\frac{1}{2}(3+1) = -2$$

$w_2=0$  意味着特征空间的点的第二个维度对分类结果没有影响。判别条件是: 如果  $x_1 > 2$ , 则该点属于  $C_+$  类; 如果  $x_1 < 2$ , 则该点属于  $C_-$  类。分类超平面如图 5-18 所示。

下面举一个计算 SVM 的例子: 在二维空间中有 4 个点, 对应的标记值是  $y$ 。四个点描述如下:

$$\begin{aligned} x_1 &= (-2, -2) & y_1 &= +1 \\ x_2 &= (-1, 1) & y_2 &= +1 \\ x_3 &= (1, 1) & y_3 &= -1 \\ x_4 &= (2, -2) & y_4 &= -1 \end{aligned}$$

在  $\alpha_i \geq 0$  的条件下求下面这个拉格朗日函数的最大值:

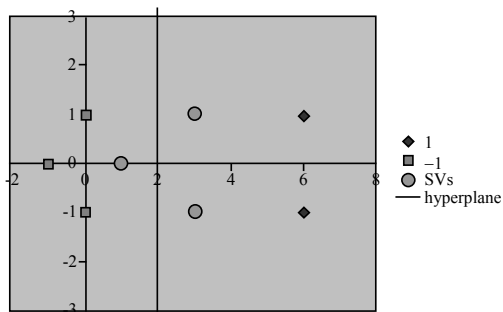


图 5-18 超平面



$$L(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$= \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 - 4 \alpha_1^2 - \alpha_2^2 - \alpha_3^2 - 4 \alpha_4^2 - 4 \alpha_1 \alpha_3 - 4 \alpha_2 \alpha_4$$

$$\text{得到: } \alpha_1=0 \quad \alpha_2=\frac{1}{2} \quad \alpha_3=\frac{1}{2} \quad \alpha_4=0$$

因此支持向量是  $x_2$  和  $x_3$ 。

$$w = \sum_i \alpha_i y_i x_i = \frac{1}{2}(x_2 - x_3) = (-1, 0)$$

$$b = y_2 - w^T x_2 = 0$$

在一维空间中，没有任何一个线性函数能解决如图 5-19 所示的划分问题（粗线和细线各代表一类数据），可见线性判别函数有一定的局限性。

如果建立一个如图 5-20 所示的二次判别函数  $g(x)=(x-a)(x-b)$ ，则可以很好地解决如图 5-19 所示的分类问题。



图 5-19 线性函数不能分类的问题

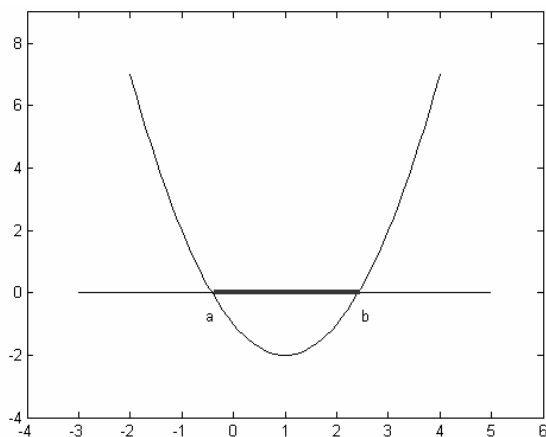


图 5-20 二次判别函数

这里  $g(x)=w_1*x^2+w_2*x+b$ 。

以前的特征空间只有一个维度，这个维度对应的值也就是  $x$  的值。现在的特征空间有两个维度，第一个维度对应的值还是  $x$  的值，第二个维度对应的值是  $x^2$  的值。

举个日常生活中的例子：假设有个十字路口。连续不停南来北往的车辆会让这个十字路口拥堵。即使有红绿灯也无法完全解决问题，但那是因为从平面交叉的二维角度来看才会无解。如果在这里建一个立交桥，车流就会变得顺畅无阻。也就是说加入“高度”这个维度，进入三维空间，问题就能得到更好的解决。

因此，可以通过核函数把特征空间映射到更高的维度，这样有可能找到更好的超平面。图 5-20 的例子是多项式核，此外常用的还有 RBF 核。

使用一个映射函数  $\Phi$  把输入空间中的数据转换成特征空间中的数据使得分类问题变成线性可分的。然后求解出最优分隔超平面。当超平面通过  $\Phi^{-1}$  映射回输入空间时，超平面就变成一个复杂的决策表面。

分类问题可以转化成相似度计算的问题。如果待分类的点和正类点之间的相似性高则可以把待分类的点分到正类，反之，如果待分类的点和负类点之间的相似性高则可以把待分类的点分到负类。相似性可以用向量间夹角的余弦，或者说两个向量的内积表示。用核函数  $K(x_i, x_j) = \Phi(x_i)^T \Phi(x_j)$  来度量两个点的相似性。

回顾前面介绍的文本分类的方法把一个文档映射成有很高维度的向量。这种方法丢失了所有的词的顺序信息，而仅仅保留了文档中的词的频率信息。字符串核（String kernel）通过从文档中提取  $k$  个前后相连的字作为特征来保留词的顺序信息。字符串核的另外一种实现方法是：首先把字符串转换成后缀树（Suffix Tree），然后构造树核（Tree Kernel）。

因为训练集中存在噪声，尤其因为训练集不能代表全部判决空间，所以正确处理相对模糊的间隔区域（margin）与全集的比例是关键。最优分类超平面是指：该分类面不但能正确分类而且使各类别的分类间隔最大。最优训练的含义是：在确定的平面分类间隔条件下，不但训练序列的判决正确率高，而且推广到全集序列的判决率也尽可能高。

SVM 最基本的方法只能实现两类分类，可以通过组合多个两类分类器来实现多类分类。

例如有  $N$  类，一种解决方法是学习  $N$  个 SVM 分类器。

SVM 1 学习 “Output==1” vs “Output != 1”

SVM 2 学习 “Output==2” vs “Output != 2”

...

SVM  $N$  学习 “Output== $N$ ” vs “Output !=  $N$ ”



计算文档属于每个类别的隶属度，然后取最大隶属度对应的类别。这个方法叫做一类对余类。取最大隶属度对应类别的实现代码如下所示：

```
//输入每个类别的隶属度组成的数组
//得到文档的所属类别 ID
private int singleCategory(double[] simRatio) {
    int catID = 0;
    double maxNum = simRatio[catID];
    for (int i = 1; i < m_nClassNum; i++) {
        if (simRatio[i] > maxNum) {
            maxNum = simRatio[i];
            catID = i;
        }
    }
    return catID;
}
```

另外介绍一种两类分类器来实现多类分类的方法，叫做错误校正输出编码（Error Correcting Output Coding）。例如取  $m = 4$  个类别。

政治、体育、商业、艺术

分配唯一的  $n$  位向量给每个类名，这里  $n > \log_2 m$ 。第  $i$  位向量作为类名  $i$  的唯一编码。 $m$  个类名组成的位矩阵记作  $C$ 。

对每列构建独立的二元分类器。这里是 10 个分类器。正类是  $C_{ij} = 1$  对应的类别；负类是  $C_{ij} = 0$  对应的类别。例如：第三个分类器把{体育、艺术}作为负类，把{政治、商业}作为正类。具体类名编码如表 5-7 所示。

表 5-7 类名编码

类 名	编 码
政治	0110110001
体育	0001111100
商业	1010101101
艺术	1000011010

通过插件分类器判断文档类别。把预测某一位值的分类器叫做插件分类器，一个插件分类器预测文档属于某个类别的子集。根据  $\{\lambda^1, \lambda^2, \dots, \lambda^n\}$  来判断文档的类别。

计算文档  $x$  的类别时，先生成一个  $n$  位的向量。

$$\lambda(x) = \{\lambda_1(x), \lambda_2(x), \dots, \lambda_n(x)\}$$

生成的位向量  $\lambda(x)$  很有可能不是  $C$  中的一行，但是可能更像某些行，也就是和某些行

的海明距离更近。把文档分类成这行对应的类别，也就是如下的公式：

$$\operatorname{argmin}_i \text{HamingDistance}(C_i, \lambda(x))$$

可以根据海明距离判断  $\lambda(x)$  和哪行最相似。

假设  $C_i$  和  $\lambda(x)$  最相似，则第  $i$  个类别作为文档  $x$  的类别。如果生成的位向量是  $(x) = \{1010111101\}$ ，则把这篇文档分到“商业”类。

自动分类的 SVM 方法接口分为训练过程接口和执行分类的接口。分类器训练过程的接口如下所示：

```
//创建一个分类器
Classifier svmClassifier = new Classifier();
//设置训练文本的路径
svmClassifier.setTrainSet("D:/train");
//设置训练输出模型的路径
svmClassifier.setModel("D:/model");
//执行训练
svmClassifier.train();
```

分类器训练好的结果写入 D:/model 目录，然后执行分类的接口如下所示：

```
//加载已经训练好的分类模型
Classifier theClassifier = new Classifier("D:/model/model.prj");
//文本分类的内容
String content = "我要买把吉他，希望是二手的，价格 2000 元以下";
//开始使用 SVM 方法分类
String catName = theClassifier.getCategoryName(content);
System.out.println("类别名称:" + catName);
```

### 5.10.5 规则方法

朴素贝叶斯和 SVM 的方法基于文档中很多单词的组合加权来判断文档的类别，但是人工无法调整从训练集中学习出来的分类模型。人工编写的规则容易理解，而且可以达到很高的精度，但是完全由人工开发和维护的规则模型代价昂贵。

关键词识别规则不是基于单词频率，而是基于某个单词有没有出现在指定的位置。例如：如果文档中出现“NBA”这个词就把这篇文档归到“体育”类。

实现文本分类规则的代码如下所示：



```

public class Rule {
    public String[] antecedent;//关键词
    public String consequent;//分类结果

    public Rule (String[] antd,String classLabel){
        antecedent = antd;
        consequent = classLabel;
    }

    /**
     * 规则是否覆盖分类文档，也就是这个文档是否满足规则的条件
     *
     * @param doc 待分类实例
     * @return 如果满足分类条件，则返回 true
     */
    public boolean covers(HashSet<String> doc){
        for(int i=0;i<antecedent.length;++i)    { //测试条件中的每个词
            if(!doc.contains(antecedent[i])) {
                return false;
            }
        }
        return true;
    }
}

```

可能会有多个规则符合同一个分类实例，这样可能产生歧义。为了解决歧义，可以给规则定义一个顺序，并从上往下应用规则。这样，规则集合就变成了决策列表。决策列表就是一个有序的规则集合。给定一个分类实例  $t$ ，按照指定的顺序应用规则，直到一个规则的模式匹配上  $t$ ，然后把实例  $t$  归到  $R$  对应的分类结果，流程如图 5-21 所示。

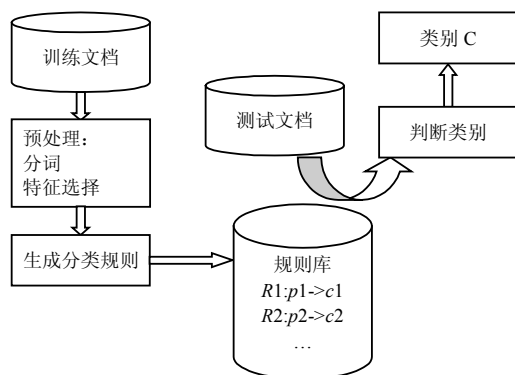


图 5-21 基于规则的文本分类流程图

根据规则对文本分类的代码如下所示：

```

/**
 * 对给定的文本进行分类
 * @param text 给定的文本
 * @return 分类结果
 */
public String classify(String text) {
    ArrayList<String> terms = splitter.getWords(text); //中文分词处理
    HashSet<String> instance = new HashSet<String>();
    instance.addAll(terms);

    for (int i = 0; i < decisionList.size(); i++) {
        Rule rule = decisionList.get(i);
        if (rule.covers(instance))
            return rule.consequent;
    }
    return "";
}

```

学习出分类模型可以看生成规则和对规则排序两个过程。

生成规则的方法说明如下：先用特征选择的方法（例如 CHI）生成每个类别的特征词。然后根据共同出现在一个句子中的特征词组合生成规则，最后验证规则的有效性。

规则可以由人工编写，可以采用搜索的方法初步检验规则的有效性。把规则的条件作为搜索词输入，如果返回的搜索结果按类别的分类统计只有一类，则说明这个规则有 100% 的精确度。例如从 <http://search.gongkong.com/SearchProduct.aspx> 搜索“变频器 AND 矢量 AND 解耦”，只返回“低压变频器”类别的产品。由此得到规则：

变频器 矢量 解耦=> 低压变频器

除了人工编写，还可以采用机器学习的方法从训练集学习出分类规则，例如 PRISM 和 RIPPER 算法。

PRISM 算法在构建每个规则后，先删除这个规则覆盖的那些文档，然后再找新规则。从高覆盖度的规则开始，通过增加更多的条件来提高它的准确度，争取最大化每个规则的准确度。当准确度是 1，或者没有更多的文档时，停止增加条件。PRISM 算法的伪代码如下所示：

```

for each class C
    initialise E to the complete instance set
    while E contains instances with class C

```



```
create empty rule R if X then C
until R is perfect (or no more attributes)?
    for each attribute A not in R, and each value v,
        consider adding A=v to R
        select A and v to maximise accuracy of p/t
        add A=v to R
remove instances covered by R from E
```

RIPPER 规则学习算法针对两类问题，选择一类作为正类，另外一类作为负类。学习分到正类的规则，把负类作为默认类别。对于多类分类问题，按类的流行程度（有多大比例的文档属于一个类别）增加的方式对类别排序，最小的类首先学习规则集合，把其他的类别看成是负类，重复让下一个最小的类别作为正类。

根据训练集对给定的规则排序就是求解指定规则集合的最优排序，使得正确标注的实例数量最大。

发现决策列表的贪心算法说明如下：每次迭代过程选择一个规则，然后把这个规则加入到决策列表的末尾，删除选择出的规则。继续这个过程直到输出所有的规则。每次选择规则时，会使用一个打分函数对每个规则打分，然后选择有最大分值的规则。

### 5.10.6 网页分类

抓取的新闻网站（如 <http://news.sina.com.cn/>）上面本来就有国际新闻和国内新闻的栏目，如果索引页能分成“国际新闻”或“国内新闻”的一种，详细页的类别参考索引页的分类，那么就可以形成一个“国际新闻”和“国内新闻”的分类训练样本库。

另外，从 <http://bj.58.com/job.shtml> 网址就可以知道它可能是和招聘相关的网页，而且可能是北京地区的网页。因此，使用 URL 地址可以快速分类网页。例如，凤凰网中新闻的 URL 地址：

[http://news.ifeng.com/mil/4/detail\\_2010\\_09/11/2489355\\_0.shtml](http://news.ifeng.com/mil/4/detail_2010_09/11/2489355_0.shtml)

URL 中包含 reading 的就分为读书类；URL 中包含 mil 的就分为军事类；URL 中包含 culture 的就分为文化类。

为了提取“<http://bj.58.com/job.shtml>”的特征词“bj”、“58”、“job”，首先按“:”、“/”、“.”切分，然后通过词干化和小写化处理，去除停用词“http”、“com”、“shtml”，最后剩下“bj”、“58”、“job”三个分类特征词。



## 5.11 拼音转换

当搜索“zhonghuarenmingongheguo”，搜索结果会提示：您是不是要找“中华人民共和国”。有了这项功能以后，用户可以直接输入拼音串搜索。在具体实现上，可以采用用户搜索的高频词和汉语拼音对照的方法来解决。

因为存在多音字的问题，所以不能仅仅只有字和音的对照表。首先准备好拼音词库，例如：

```
jvnguanzheng:军官证:0
shuangchunyin:双唇音:0
bolinqiang:柏林墙:0
jiaoyi:交友:0
zhanhexiang:战河乡:0
shouzhi:手织:0
liuyixiang:柳驿乡:0
anxiang:安享:0
moganshanzhen:莫干山镇:0
zuochou:柞绸:0
wuxia:无暇:7
meileike:梅雷克:0
```

这里的词典格式是：每个词一行，第一列是音，第二列是词，第三列是词的频率。

因为同一个拼音串的对应的候选汉字串较多，需要借助于上下文才能选出更有意义的候选串，这里先建立词的二元模型。可以从搜索日志中学习二元连接词典 BigramDict.txt。

最后调用 Converter 把拼音转换成汉字。

```
Convertor convertor = new Convertor();
String yin = "dianji";
word = convertor.find(yin);
```

执行后返回 word 中返回结果是“电机”。



## 5.12 概念搜索

一个词可能有好几个意思。例如“地道”有两个意思，作为名词时表示“在地面下掘成的交通坑道”，作为形容词时表示“纯粹的，真正的”。另外有些词可以表示同样的意思，例如“西红柿”和“番茄”是 synonym，“招商行”是“招商银行”的简称，也算 synonym。

可以从网络挖掘出 synonym 词库。例如有下面一些链接到同一个网站的标签。

```
<a href="http://www.cmbchina.com/" target="_blank">中国招商银行</a>
<a href="http://www.cmbchina.com/" target="_blank">招行</a>
<a href="http://www.cmbchina.com/" target="_blank">招商银行</a>
```

从上面的链接中可以提取出 synonym：

中国招商银行 招行 招商银行

还可以从语料库挖掘 synonym。先找这个词与哪些词有比较强的关联，得到一个向量，所有的词通过向量比较就能知道 synonym 程度。

另外可以通过句子模版发现 synonym。从 Google 搜索“大豆 又称”，可以发现“大豆（又称黄豆）”，这样可以找到“大豆”的 synonym “黄豆”。

我们这里通过 synonym 搜索来尝试语义扩展的搜索。具体的实现方法是通过查询扩展的方式。当用户输入“计算机”搜索的同时，程序通过查找 synonym 词库，按照“计算机”、“电脑”、“微机”等多个 synonym 查找。当用户输入“轿车”的同时，也能按照“奥迪”、“奔驰”等进行下位词扩展。

简单的做法是把 synonym 保存在一个 HashMap 中。如果 synonym 词表很大，可以把 synonym 保存在一个索引库中。

SynonymEngine 是一个很简单的接口，输入一个词，返回它的 synonym。

```
public interface SynonymEngine {
    String[] getSynonyms(String s) throws IOException;
}
```

DexSynonymEngine 使用一个简单的存储在散列表中的 synonym 实现。

```

public class DexSynonymEngine implements SynonymEngine {
    //词和对应的同义词数组
    private static Map<String, String[]> map = new HashMap<String,
    String[]>();

    static {
        //数字同义
        map.put("1", new String[] { "一" });
        map.put("2", new String[] { "二" });
        map.put("3", new String[] { "三" });
        map.put("4", new String[] { "四" });
        map.put("5", new String[] { "五" });
        map.put("6", new String[] { "六" });
        map.put("7", new String[] { "七" });
        map.put("8", new String[] { "八" });
        map.put("9", new String[] { "九" });
        map.put("10", new String[] { "十" });

        //日期同义
        map.put("非周末", new String[] { "周一", "周二", "周三", "周四", "周五" });
        map.put("周末", new String[] { "周六", "周日" });

        //词同义
        map.put("西红柿", new String[] { "番茄" });
        map.put("黄豆", new String[] { "大豆" });
    }

    public String[] getSynonyms(String word) throws IOException {
        return map.get(word);
    }
}

```

增加一个 `SynonymFilter`。

```

public class SynonymFilter extends TokenFilter {
    public static final String TOKEN_TYPE_SYNONYM = "SYNONYM";

    private Stack synonymStack;
    private SynonymEngine engine;
    private TermAttribute termAttr;
    private AttributeSource save;

    public SynonymFilter(TokenStream in, SynonymEngine engine) {
        super(in);
        synonymStack = new Stack(); //同义词缓存
    }
}

```



```
termAttr = (TermAttribute) addAttribute(TermAttribute.class);
save = in.cloneAttributes();
this.engine = engine;
}

public boolean incrementToken() throws IOException {
    //弹出缓存的同义词
    if (synonymStack.size() > 0) {
        State syn = (State) synonymStack.pop();
        restoreState(syn);
        return true;
    }
    //读下一个词
    if (!input.incrementToken())
        return false;

    //把当前词的同义词压入堆栈
    addAliasesToStack();

    //返回当前的词
    return true;
}

private void addAliasesToStack() throws IOException {
    String[] synonyms = engine.getSynonyms(termAttr.term()); //查询同义词
    if (synonyms == null) return;
    State current = captureState();

    //把同义词压入堆栈
    for (int i = 0; i < synonyms.length; i++) {
        save.restoreState(current);
        AnalyzerUtils.setTerm(save, synonyms[i]);
        AnalyzerUtils.setType(save, TOKEN_TYPE_SYNONYM);
        //设置位置增量为 0
        AnalyzerUtils.setPositionIncrement(save, 0);
        synonymStack.push(save.captureState());
    }
}
}
```

然后在 `SynonymAnalyzer` 中使用它。

```
public class SynonymAnalyzer extends Analyzer {
    private SynonymEngine engine; //保存了一个词的同义词
```

```

public SynonymAnalyzer(SynonymEngine engine) {
    this.engine = engine;
}

public TokenStream tokenStream(String fieldName, Reader reader) {
    TokenStream result = new CnTokenizer(reader); //先分词
    result = new SingleFilter(result); //再拆分成单字
    result = new SynonymFilter(result, engine); //在 TokenStream 中
    //增加同义词
    return result;
}
}

```

索引时使用 `SynonymAnalyzer` 把同义词扩展放到索引库，在搜索时使用下面这个 `CnAnalyzer`，不做同义词扩展。

```

public class CnAnalyzer extends Analyzer {
    public TokenStream tokenStream(String fieldName, Reader reader) {
        TokenStream stream = new CnTokenizer(reader);
        stream = new SingleFilter(stream);
        return stream;
    }
}

```

实现这个功能的基本步骤说明如下。

- ① 准备语义词库。
- ② 把语义词库转换成同义词索引库。
- ③ 在 `SynonymAnalyzer` 中使用同义词索引库。

英文的同义词词库最著名的是 `WordNet`，它的介绍和下载地址为 <http://wordnet.princeton.edu>。中文方面，“同义词词林”和 `HowNet` 都有现成的同义词库。以“同义词词林”为例，它用树形结构表示了词的同义和上下位关系。

下面是同义词的例子：

Bp31B 表

Bp31B01= 表 手表

Bp31B02= 马表 跑表 停表





Bp31B03= 怀表 挂表  
Bp31B04= 防水表 游泳表  
Bp31B05= 表针 指针  
Bp31B06= 表盘 表面  
Bp31B07# 夜光表 秒表 电子表 日历表 自动表  
Bp31B08= LONGINES 浪琴  
Bp31B09= Rema 瑞尔玛  
Bp31B10= ROSSINI 罗西尼  
Bp31B11= SEIKO 精工  
Bp31B12= SUUNTO 松拓  
Bp31B13= Tissot 天梭  
Bp31B14= CASIO 卡西欧

整理出这样的词表后，我们通过下面的程序转换成 WordNet 的 prolog 同义词数据库 wn\_s.pl 格式：

```
s(synset_id,w_num,'word',ss_type,sense_number,tag_count).
```

第一列是语义编码，同一个语义有唯一的语义编码列；第二列是单词编号；第三列是单词本身；第四列是词性；第五列代表该词第几个语义；第六列是该词在语料库中出现的频率。例如：

```
s(100006026,1,'person',n,1,7229).  
s(100006026,2,'individual',n,1,51).  
s(100006026,3,'someone',n,1,17).  
s(100006026,4,'somebody',n,1,0).  
s(100006026,5,'mortal',n,1,2).  
s(100006026,6,'human',n,1,7).  
s(100006026,7,'soul',n,2,6).
```

同义词词林按照该格式整理如下：

```
s(Dk02A12,1,'中学',n,1,0).  
s(Dk02A12,2,'国学',n,1,0).  
s(Dk02A12,3,'旧学',n,1,0).
```

这个辞典格式还没有词性信息，而词典中的词有词性却没有词义信息。可以借助词典中的词性猜测同义词词林中义项的词性。

下面我们给基本词典（coreDict.txt 文件）标注语义。基本词典格式是：

词：词性：词频：拼音：语义 1，语义 2

例如：

一举一动：26880:2:yijuyidong:Di20B01

实现过程说明如下。

### ① 给同义词词林的每个义项增加词性。

为了更好地消除歧义，可以给每个义项增加词性。例如：“行”作为名词用的时候可能是“行业”的意思，语义编码是 Di18B01；“行”作为量词用的时候可能是“队列”的意思，语义编码是 Dd07B01。

可以参考基本词典中对应的每个词的词性，然后统计最有可能的词性。

② 给基本词典中词频大于 1 的常用词加上义项到同义词词林。100 这样的伪词频不用管，只需要把每个常用词分类到二级子类即可。例如归类到“Be”这类就可以。

Be 地 貌 地 形

全部内容如下：

Be01 陆地 原野 沙漠 陆空

Be01A01= 陆地 大陆 陆 洲 地 陆上 次大陆 新大陆 大洲 沂

...

Be03 滩 岸

Be04A50= 恒山 北岳

...

Be05 海洋 江河 溪涧 濒海

Be05A01= 海洋 大海 大洋 五洋 沧海 海洋 溟 瀛 瀛海 沧溟 重溟 大壑 水宗  
海域 浅海 汪洋大海 深海

Be05A02= 远洋 重洋

Be05A03= 洋流 海流



Be05A04= 大陆架 陆棚 大陆棚 陆架 大陆坡

Be05A05= 内海 内陆海 陆海

Be05A06= 海岭 海脊

Be05A07# 海湾 海峡 海沟 海穹 海床

Be05A08= 国际海域 公海

最后可以采用字面搜索分类的方法给未标注语义的词标注上语义。

利用 Lucene 筛选最相关词时，应先对同义词词林中所有的词按字创建索引，索引库分为两列，第一列是词本身，第二列是每个词所属的类别，然后把要归类的词作为搜索关键词查找同义词索引库。

例如“海勒斯台高勒河”不在已有的同义词表中。看起来既有“海”又有“河”，所以 Be 类的词匹配的可能比较多。搜索后统计匹配上的词的类别，将匹配上结果最多的类作为“海勒斯台高勒河”的类别。

同义词词林格式到 WordNet 格式实现转换的代码如下所示：

```
public static void CL2WordNet(String sSourceFile,String save_filename,
String dic_file){
    DicCore dic = new DicCore(dic_file);
    BufferedReader fpSource = null;
    BufferedWriter output=null;
    String sLine;
    StringTokenizer st = null;
    String number= null;

    fpSource= new BufferedReader(new FileReader(sSourceFile));
    output = new BufferedWriter(new FileWriter(save_filename));
    while( (sLine = fpSource.readLine()) != null ) {
        if(sLine.length()>9 )    {
            int pos = sLine.indexOf('=');
            if (pos<=0)    {
                continue;
            }
            number = sLine.substring(0,pos);
            st = new StringTokenizer(sLine.substring(pos + 1 ).trim(),"
\\t\\n\\r");
            String word;
            int count = 0;
            String wordPOS = "n";
```

```

while(st.hasMoreElements()) {
    count++;
    word = st.nextToken();
    POSQueue posQ = dic.get(word);
    if(posQ==null) {
        System.out.print(word+": "+wordPOS+"\n");
    }
    else if (posQ.size()==1) {
        wordPOS = DicCore.gePOSName(posQ.getHead().item.
            nPOS);
    }

    output.write("s("+number+", "+count+", '"+word+"', "+wordPOS+", 1, 0). "
        +"\n");
}
}
}
output.close();
fpSource.close();
}

```

然后通过程序 Syns2Index，把 wn\_s\_cn.pl 转换成 Lucene 同义词索引库。通过 Luke 索引工具可以查看到索引结构是一个 word 值对应的多个 syn 同义词，如图 5-22 所示。

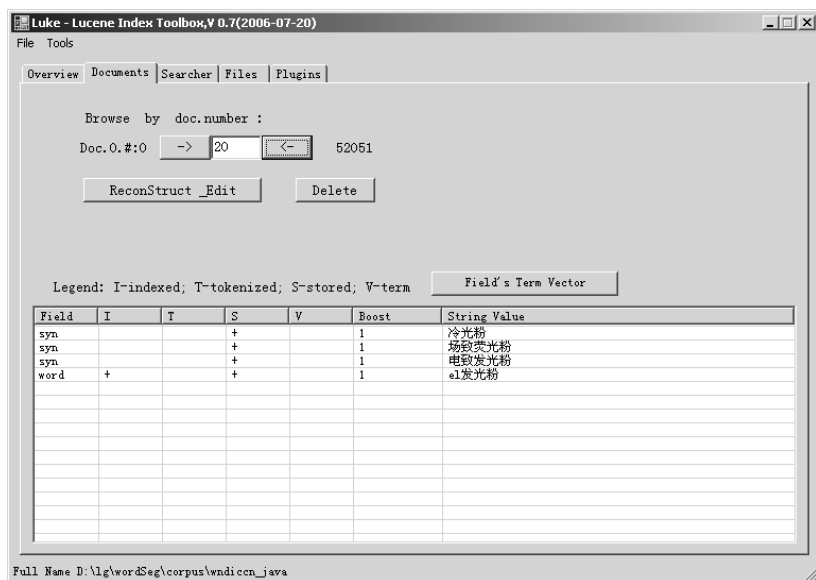


图 5-22 同义词索引库



最后我们通过 `SynonymAnalyzer` 来调用这个同义词索引库实现同义词扩展查找。但是 `SynonymAnalyzer` 只是简单地通过词本身来扩展同义词，这样并不一定准确，尤其是对词义很多的词来说。Lucene 4.0 的灵活索引出来以前，不允许在索引中任意存储信息。但是 Lucene 的 2.2 版本以后支持把对词的额外描述存储在 `Payload` 属性中，所以可以把一个词的语义编码存储在词的 `Payload` 属性中。

可以思考一下动态同义词的实现。搜索星期日的时候同时返回下一个最近的日期，例如离 2010 年 11 月 5 日最近的一个星期日是 2010 年 11 月 7 日。



## 5.13 多语言搜索

Lucene 支持多种世界上流行的语言，如表 5-8 所示。

表 5-8 Lucene 对多种语言的支持

语 言	支 持 包
俄语	Org.apache.lucene.analysis.ru.RussianAnalyzer <a href="http://code.google.com/p/russianmorphology/">http://code.google.com/p/russianmorphology/</a>
法语	org.apache.lucene.analysis.fr.FrenchAnalyzer
日语	<a href="http://sourceforge.jp/projects/chasen-legacy/">http://sourceforge.jp/projects/chasen-legacy/</a>
韩语	<a href="http://sourceforge.net/projects/lucenekorean/">http://sourceforge.net/projects/lucenekorean/</a>
简体中文	org.apache.lucene.analysis.cjk.CJKAnalyzer org.apache.lucene.analysis.cn.ChineseAnalyzer
德语	Org.apache.lucene.analysis.de.GermanAnalyzer
泰语	org.apache.lucene.analysis.th.ThaiAnalyzer

<https://issues.apache.org/jira/browse/LUCENE-2206> 包含了处理丹麦语、荷兰语、英语、芬兰语、法语、德语、匈牙利语、意大利语、挪威语、俄语、西班牙语和瑞典语的停用词表和词干化功能。

`SnowballAnalyzer` 举两个词干化的例子：

```
Analyzer analyzer = new SnowballAnalyzer("Spanish"); // 西班牙文
Analyzer analyzer = new SnowballAnalyzer("Portuguese"); // 葡萄牙文
```

新中国建立以前曾拥有和使用本民族文字的，有藏、蒙古、维吾尔、哈萨克、柯尔克孜、朝鲜、傣、彝、俄罗斯、苗、纳西、水、拉祜、景颇、锡伯等民族。新中国成立以来，国家为促进少数民族文化教育事业的发展，帮助一些少数民族改进和创制了文字，目前，

我国已正式使用和经国家批准推行的少数民族文字有蒙古文、藏文、维吾尔文、朝鲜文、壮文等 19 种。现在世界各国所用的文字多数是拼音文字，我国的藏文、蒙文、维吾尔文等也都是拼音文字。



## 5.14 跨语言搜索

搜索内容中有繁体的内容。能想办法进行简繁转换吗？比如说搜索简体可以出现含繁体字内容的结果，反之亦然。

可以的，首先要有简繁互相转换的模块，然后可以在索引库中对同样的数据建立两列。一列是简体，还有一列是繁体。搜索用户界面也可以做成可在多种语言之间切换。

因为简体字和繁体字之间的对应关系是“一对多”的关系。所以从繁体转换到简体比较简单，只需要一个如下的汉字对照表，然后查表即可。

窩:窝  
箋:笺  
箏:筝  
箇:个  
綰:绾  
綜:综  
綽:绰  
綾:绫  
綠:绿  
緊:紧  
綴:缀  
網:网

简体转换到繁体因为存在一个简体中文的一个字在繁体中对应多个字的情况，所以实现起来比繁转简要复杂。例如“出来演落幕的一出戏”翻译成繁体是“出來演落幕的一齣戲”。

简体转繁体的过程是：简体文字做分词，通过词级别的对照转换到繁体。如下是一个简繁词语对照表：

完好无恙:完好無恙  
完备:完備  
完备:完備



完好无缺：完好無缺  
完好无损：完好無損  
宏伟：宏偉  
完璧归赵：完璧歸趙

最后通过简繁转换类 `SimpTradConv` 实现简体转换成繁体：

```
String result = SimpTradConv.trad("出来演落幕的一出戏");  
System.out.println("the result:"+result);
```

转换的结果是“出來演落幕的一齣戲”。

如果搜索内容中有英语或者日语的内容，则需要使用机器翻译技术把搜索内容翻译过来。最简单的方法是使用逐个查词表的方式把外语句子翻译为中文。以英文翻译成中文为例：

```
//英文到中文的对照词表  
HashMap<String,String> ecMap = new HashMap<String,String>();  
ecMap.put("John", "约翰"); //放入一个键/值对  
ecMap.put("loves", "爱");  
ecMap.put("Mary", "玛丽");  
  
String english = "John loves Mary"; //要翻译的英文句子  
  
//用空格分割英文句子  
StringTokenizer tokenizer = new StringTokenizer(english);  
while(tokenizer.hasMoreElements()){ //有更多的词没遍历完  
    System.out.print(ecMap.get(tokenizer.nextToken())); //输出：约翰爱玛丽  
}
```

“programming”翻译成“编程”，但是在“dynamic programming”中翻译成“动态规划”。用最大长度匹配算法解决这个问题。英译汉最大长度匹配方法的翻译过程如图 5-23 所示。

机器翻译更大的挑战来自于语序调整问题。例如，英文往往把定语放在被修饰词后面。“history of php”翻译成“PHP 的历史”。使用依存树来调整语序。首先得到英文的依存树，然后把英文树转换成为中文树，最后把中文树转换成字符串的方式得到翻译结果。实现代码如下：

```
String enSent = "history of php";  
  
//得到英文词序列  
ArrayList<WordToken> tokens = ProbMT.getWords(enSent);  
//根据词序列得到英语依存树
```

```

ArrayList<EnDepTree> depTrees = TreeConverter.getDepTree(tokens);

for (EnDepTree t : depTrees) {
    //英文依存树转换成中文依存树，实现句子结构的转换，以及英文词到中文词的转换
    CnDepTree targetTree = t.convert();
    String sent = targetTree.toSentence(); //输出中文依存树
    System.out.println(sent); //输出:PHP 的历史
}

```

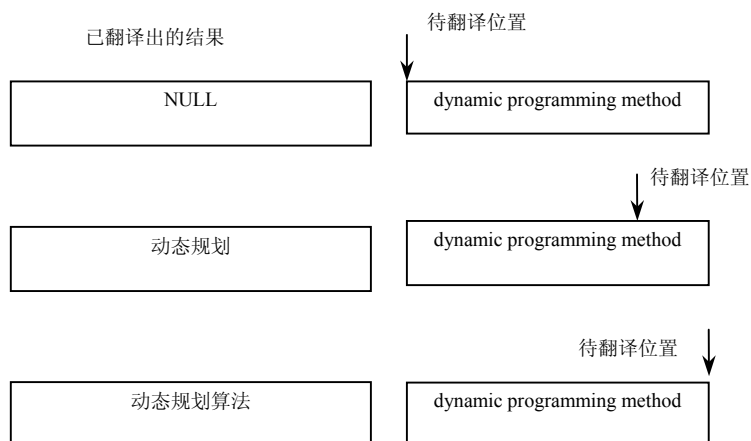


图 5-23 最大长度匹配翻译过程



## 5.15 情感识别

可以结合核磁共振，通过对大脑中的兴奋区域成像来分析“喜欢”或“厌恶”等情感倾向。这里讨论的情感识别又叫做文本倾向性分析，英文叫做 *sentiment analysis* 或 *Opinion mining*。基本的目标就是实现区分出正面、负面或者中性，这叫做极性分类（*polarity classification*）。可以按好恶程度分出更多的级别，例如，1~5 星级，这叫做星级评分（*multi-way scale*）。

有文档级别的情感识别，例如对某个电影或酒店的评论自动分类出极性或者星级，这样区分出好评和差评。也许想进一步对好在哪里、差在何处做更细致的分析，所以出现了更细粒度的基于特征的情感识别，例如区分出对手机屏幕或者照相机画质的评价。为了准确地识别极性，可以考虑对文本的主客观语句分类，提取出  $n$  个最主观的句子来概括整个评论的褒贬倾向。从技术上来说，就是从主客观混合文本语料中抽取表示主观性的文本。





为了实现基于特征的情感识别，需要从上下文提取出评价的对象。需要提取描述对象的特征，然后判断倾向性描述在每个特征上的极性。“特征”一词在这里既表示描述对象的组成也表示属性。

特征抽取是获得关于主题某一方面的具体描述，如汽车的油耗与操控性、数码相机的电池寿命。和信息抽取相比，情感分析中的特征抽取更加自由，因为获得的结果不要求是结构化的。在某些应用中，特征抽取比情感取向判断更加重要，因为需要关注用户的具体意见。例如对某款照相机的评价统计：

照相机：

褒义：125 <独立的评价句子>

贬义：7 <独立的评价句子>

特征：画质

褒义：123 <独立的评价句子>

贬义：6 <独立的评价句子>

特征：大小

褒义：82 <独立的评价句子>

贬义：10 <独立的评价句子>

对事物的观点有直接观点和对比观点两种。

- 直接观点（direct opinion）：例如，这款相机的画质的确有点烂。
- 对比观点（comparative opinion）：例如，这款相机的画质比 Camera-x 好。进行这类情感分析时，首先要确定观点的目标对象是谁。在这个例子中需要用到指代消解确定这款相机指哪款照相机。

有时候，作者把情感和事实一起来表达，如“3 寸的液晶显示屏取景非常细致清晰”。情感和具体的特征是分不开的。

除了这些经典的问题外，在针对社会媒体的情感分析中，我们面临更多的挑战。例如，并非所有的与主题相关的用户为中心的内容都是重要的，只有其中少部分引起关注和讨论，甚至进而影响其他用户的观念和行为。因此，评估它们的影响力和预测它们是否得到关注具有重要的应用价值。

除此以外，不合理地利用社会媒体的影响力也值得我们关注。制造事端打击竞争对手，

恶作剧心理造谣生事，收受商家好处为特定产品夸大宣传都是典型的误导公众行为。

首先从文本中抽取描述对象的特征。例如，针对汽车的用户体验信息，关于操控性、舒适性、油耗、内饰、配置等方面的评价被分别抽取列出，因此可以收集到不同用户关于同一特征的描述并在不同品牌、不同时间段、不同用户群的范围内统计加以比较评估，这样的数据能直接地、准确地反映用户的消费情况和市场反应。再次，需要评估一个用户言论的内在价值和预测将来的关注度。从实务操作上来说，有些重要的言论和事件在几个小时内就会引起广泛的关注。相关的厂家可以及时发现和跟进这种对其产品销售和品牌形象具有重要影响的言论。

为了获取标注好的文本倾向，可以从评论网站（比如豆瓣网、卓越、携程等）抓取所有的评论，这些评论用星级评价来代表褒贬度。

常见的具有语义倾向词语的词性及示例如表 5-9 所示。

表 5-9 有语义倾向的词语表

词 性 编 码	词 性	示 例
a	形容词	美丽、丑陋
n	名词	英雄、熊市、粉丝、书籍
v	动词	发扬、贬低
d	副词	昂然、暗地
i	成语	宾至如归、叶公好龙
p	习惯语	双喜临门、顺竿爬

事实上，对一篇文章而言，它表达的情感的正面或负面是通过主观语句体现出来的，如“产品质量好！”。但是像“它的售价刚好是¥50 元！”这样的客观语句，虽然有“好”这一特征词，但并不表达任何情感。如果能区分一篇文章中的主观语句和客观语句，只对主观语句进行特征选择，会对分类的准确率有很大提高。

观点搜索系统使得用户能够查找关于一个对象的评价观点。典型的观点搜索查询包括以下两种类型。

- 搜索关于一个特定对象或对象特征的观点。搜索用户只要简单给出对象和/或对象的特征即可。
- 搜索一个人或组织关于一个特定对象或者对象特征的观点。用户需要给出观点拥有者的名字和对象的名字。



判断用户的情感取向（polarity）是喜欢、不喜欢还是中性的。通过对大量用户的感情取向进行统计，我们可以了解用户对特定产品的好恶，甚至对具体的某个特征（如数码相机的镜头、电池寿命等）作出直接的判断和比较。

开源项目 LingPipe (<http://alias-i.com/lingpipe>) 包含了情感识别的实现。LingPipe 从主观混合文本语料中抽取表示主观性的文本，可以把电影评论分成正面评论和负面评价。LingPipe 主要实现了两种分类问题：

- 主观（情感）句和客观句识别；
- 正面（喜欢）或负面（不喜欢）评价。

近年来，基于情感的文本分类逐渐被应用到更多的领域中。例如，微软公司开发的商业智能系统 Pulse，它能够从大量的评论文本数据中，利用文本聚类技术提取出用户对产品细节的看法；产品信息反馈系统 Opinion Observer 利用网络上丰富的顾客评论资源，对评论的主观内容进行分析处理，提取产品各个特征及消费者对其的评价，并给出一个可视化结果。

### 5.15.1 确定词语的褒贬倾向

在词汇的褒贬计算时，会遇到如下问题：如何发现以及判断潜在的褒贬新词。要不断地扩充我们的褒贬词库，这样才能够使后续的判断尽可能地准确。通常一个小的褒贬词库在词汇的覆盖程度上并不尽如人意，但如果要穷尽所有的褒贬词汇也非易事，如何去发掘潜在的褒贬词汇，是我们亟待解决的难题；对于一些同义词，它们的褒贬性可能相反（如“宽恕”和“姑息”），我们可以根据现有的褒贬词库和同义词库进行同义词拓展，确定这些极性相反、词义相同的词汇的褒贬。

以下方法不仅能够分析出词的褒贬性，还能够给出该词的褒贬强度。而且，对于同义词的褒贬的扩展也具有有一些效果。具体步骤说明如下。

- ① 我们从网络以及现有的褒贬词典中收集出一定数量的褒贬词汇（数量 $\geq 10000$ ）作为种子词库。
- ② 对该词库进行词频统计，分别计算出每个单字在褒贬词库中的频率，根据公式计算出每个单字的褒贬性。
- ③ 最终根据公式计算出每个词汇的褒贬性。

具体公式如下：

$$P_{ci} = \frac{fp_{ci}}{fp_{ci} + fn_{ci}}$$

$$N_{ci} = \frac{fn_{ci}}{fp_{ci} + fn_{ci}}$$

其中， $fp_{ci}$  代表字  $c_i$  在褒义词库中的词频， $fn_{ci}$  代表  $c_i$  在贬义词库中的词频， $P_{ci}$  和  $N_{ci}$  分别表示该字作为褒义词时的权重和贬义词时的权重。

由于褒贬词库在数量上并不一定一致，我们对上述公式修正如下：

$$P_{ci} = \frac{fp_{ci} / \sum_{j=1}^n fp_{cj}}{fp_{ci} / \sum_{j=1}^n fp_{cj} + fn_{ci} / \sum_{j=1}^m fn_{cj}}$$

$$N_{ci} = \frac{fn_{ci} / \sum_{j=1}^m fn_{cj}}{fp_{ci} / \sum_{j=1}^n fp_{cj} + fn_{ci} / \sum_{j=1}^m fn_{cj}}$$

其中， $n$  和  $m$  分别代表褒贬词库中不同字符的个数。

$$S_{ci} = P_{ci} - N_{ci}$$

上式代表字  $C_i$  的褒贬倾向。

对于由  $p$  个字符  $C_1, C_2, \dots, C_p$  构成的词语  $w$ ，其褒贬倾向  $S_w$  定义如下：

$$S_w = \frac{1}{p} \times \sum_{j=1}^p S_{cj}$$

下一个问题是，如何得到这个要判断的词语呢？

可以从搜索引擎中搜索“褒义词”，然后把所有带引号的词都找出来。例如“保守”是个褒义词，因为搜索结果中有这个带引号的褒义词。再例如一个搜索结果条目“雷”是时尚界褒义词，所以能自动发现“雷”这个褒义词。



先标注语料，然后通过半监督的学习方法来提取出一些固定的模式，并扩展出新的评价词语和评价对象。

### 5.15.2 实现情感识别

识别句子的极性与星级评分的流程说明如下。

- ① 关键词匹配。
- ② 模板提取。
- ③ 模板匹配。
- ④ 计算极性与星级评分。

将词语分为以下 5 类：

- ① 直接能表达出褒贬倾向的词汇，包括一些名词、形容词、副词和动词，例如：精彩、荒诞。
- ② 表示程度的副词，例如：很、非常。
- ③ 否定词，例如：不、没有。
- ④ 表示转折的连词，例如：但是、却。
- ⑤ 某些合成词，即按分词的结果拆开单独看不带情感，但是整体带有情感倾向的词组。例如：创世纪，分词系统将它分成两个词，这两个词分别出现并不带有褒贬倾向，而当同时出现时，则带有一定的褒义倾向。这样的词还有“载入史册”等。

设计标注格式为：用[a,c,d,n,v,p,i]表示词性；用[1,2,3,4,5]表示类别；用[+,-,#]表示极性（褒贬性）。用[1,2,3,4,5]表示程度。

原始文本是：这部电影很精彩。

分词结果是：这/r 部/q 电影/n 很/d2 精彩/a1。/w

标注结果是：这/r 部/q 电影/n 很/d2#2 精彩/a1+2。/w

其中，很/d2#2 表示程度副词，本身不具有褒贬性，对于褒贬性的影响因子为 2。而精彩/a1+2 则表示形容词，具有褒义情感，情感程度为 2。

匹配模板，得到关键词序列：很/d2#2 精彩/a1+2。

在模板匹配成功之后，需要根据一定的规则计算出整句文本的褒贬倾向。这个规则的设定需要在一定程度上体现出语法规则，否则将很容易导致计算出的整个语句的情感倾向错误。例如，程度副词既可能出现在其中心词的左侧，也可能出现在其中心词的右侧（“很好”，“好得很”）。本系统文本褒贬倾向计算规则设定如下：

- ① 根据模板从文本中取出所有模板成分对应的词，去掉不相关的词，组成一个序列。
- ② 第一遍扫描序列，找到所有程度副词（类别为 2），将其程度值乘到模板中离其最近的一个 1 类词的程度值上（考虑到副词可能位于其中心词的前面或者后面，所以这里的“最近”是前后双向查找，同时由于副词在前的情况比较多，所以前向查找的优先级高）。具体的处理是标注程度为 3 的因子为 1.5；程度为 2 的因子为 1；程度为 1 的因子为 0.5。
- ③ 第二遍扫描序列，找到所有否定词（类别为 3），将其往后碰到的第一个 1 类词的褒贬性取反。
- ④ 第三遍扫描序列，以转折词为单位将序列分成几个小部分，对每个小部分累加其 1 类词的褒贬倾向值，然后按转折词类型的不同乘以转折词相应的权值。让步型如“虽然”，对应部分要减弱；转折型如“但是”，对应部分要加强。最后各部分相加得到文本的褒贬倾向值。计算“这部电影很精彩”得到的褒贬倾向值为 2，即最终判定为褒义评论。



## 5.16 本章小结

本章介绍的自然语言处理技术用机器学习的方法处理文本信息，让搜索引擎体现了基本的智能。

基于上下文无关文法（Probabilistic Context-Free Grammars, PCFG）的句法分析方法一直是该领域研究的主流，但 PCFG 存在的一个重要问题是语法缺少对词汇的敏感性。因此，对 PCFG 模型的改进可以分为以下两种：引入词汇化信息和扩展非终结符标记，即目前最常见的两大类句法分析模型：词汇化模型和非词汇化模型。在词汇化模型中，词汇信息在



训练语法规则模型时起主导作用，而非词汇化模型则利用非终结符的潜在信息。

文本分类的方法除了支持向量机的方法和基于规则的方法以外，还有隐含狄利克雷分配（Latent Dirichlet Allocation, LDA）等。

Hoad 和 Zobel 提供了一份语义指纹的综述“Methods for Identifying Versioned and Plagiarised Documents”和一种用于近似重复检测的基于词的相似性度量方法。他们的评估主要集中在查询文件的不同版本和抄袭文件。Bernstein 和 Zobel 在论文“Accurate discovery of co-derivative documents via duplicate text detection”中描述了一种通过全指纹来寻找来源相同文件的技术。Bernstein 和 Zobel 在论文“Redundant documents and search effectiveness”中研究了重复对检索效果的影响。他们展示了一个 TREC 数据集中 15% 的相关文件是冗余的，冗余文档对搜索结果集有显著的影响。

Henzinger 在论文“Finding near-duplicate web pages: a large-scale evaluation of algorithms”中描述了一种对于 Web 近似重复检测的大规模评估。这篇论文比较了 shingling 算法和 Simhash 算法这两种技术。Henzinger 的研究使用了 16 亿个网页，他的结果表明，对于同一个站点上的冗余页面，这两个算法效果都不好，因为相同站点让网页看上去很类似。当网页在不同站点的时候，Simhash 算法的精度达到了 50%，而 Broder 算法只有 38%。



## 第 6 章

# Lucene 原理与应用

Lucene 源码分为核心包和外围功能包。核心包实现搜索功能，外围功能包实现高亮显示等辅助功能。Lucene 源码的核心包中共包括 7 个子包，每个包完成特定的功能。

最基本的是索引管理包（`org.apache.lucene.index`）和检索管理包（`org.apache.lucene.search`）；索引管理包实现索引建立、删除等。检索管理包根据查询条件检索得到结果。

索引管理包调用数据存储管理包（`org.apache.lucene.store`），主要包括一些底层的 I/O 操作，同时也会调用一些公用的算法类（`org.apache.lucene.util`）。编码管理包（`org.apache.lucene.codecs`）用于方便自定义索引的编码和结构。文档结构包（`org.apache.lucene.document`）用于描述索引存储时的文档结构管理，类似于关系型数据库的表结构。

查询分析器包（`org.apache.lucene.queryParser`）实现查询语法，支持关键词间的运算，如与、或、非等。语言分析器（`org.apache.lucene.analysis`）主要用于对放入索引的文档和查询词切词，支持中文主要是扩展此类。

往 Lucene 中放的是文档，查询的是词，查询返回的也是文档。





## 6.1 Lucene 深入介绍

为了方便索引大量的文档，Lucene 中的一个索引分成若干个子索引，叫做段(segment)。段中包含了一些可搜索的文档。在给定的段中可以快速遍历任何给定索引词在所有文档中出现的频率和位置。IndexWriter 收集在 RAM 中的多个文档，然后在某个时间点把这些文档写入一个新的段，写入点可以通过 Lucene 内部的配置类或者外部程序控制。然后这些文档组成的段会保持不动，直到 Lucene 把它合并进入大的段。MergePolicy 控制 Lucene 如何合并段。

### 6.1.1 常用查询对象

Lucene 中一些常用的查询对象如表 6-1 所示。

表 6-1 Lucene 中的查询对象表

Query	说 明	用 法
TermQuery	最基本的词条查询	查询不切分的字段
BooleanQuery	布尔逻辑查询	组合条件查询
PhraseQuery	短语匹配查询	要求精确匹配的查询
SpanQuery	匹配位置相关的查询 (跨度查询)	字词混合查询
FieldScoreQuery	函数查询(通过数字型的字段影响排序结果)	时间加权排序
AutomatonQuery	有限状态查询(Finite-State Query)支持“dogs~”这样的模糊查询语法	查询扩展、拼写检查

BooleanQuery 和 PhraseQuery 这样的复杂查询会分解成简单的词查询来执行。

### 6.1.2 查询语法与解析

- 一个短句查询可以用双引号括起来，这样只有精确匹配该短语的文档才会匹配查询条件出来。比如：搜索“上海世博会”将会只出现包含连续出现过“上海世博会”的文档。“上海世博会”比“上海 AND 世博会”这样的查询返回的结果更少。
- 使用^表示加权。例如搜索“solr^4 lucene”。
- 修饰符 + - NOT。例如搜索“+solr lucene”。
- 布尔操作符 OR AND。例如搜索“(solr OR lucene) AND user”。

- 按域查询。一个字段名后面跟冒号，再加上要搜索的词语或短句，就可以把搜索条件限制在该字段。例如搜索“title:NBA”，匹配标题包含 NBA 的文档。

QueryParser 将输入查询字符串解析为 Lucene 的 Query 对象，如图 6-1 所示。QueryParser 是使用 JavaCC 生成的词法解析器。JavaCC (Java Compiler Compiler) 是一个类似 YACC 的工具软件。JavaCC 可以用于词法分析和语法分析。JavaCC 根据输入文件中定义的语法生成实际用于词法分析和语法分析的程序源代码。在 Lucene 中，QueryParser.jj 定义了查询语法。

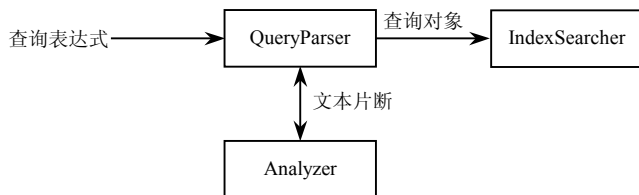


图 6-1 使用查询表达式搜索

词法解析器是如何工作的呢？首先把用户输入定义的 Token 转换为正规文法等价的形式，然后把正规文法转换成 NFA，再把 NFA 转换成 DFA，最后生成代码模拟 DFA。

要改变查询分析器的某一部分，例如查询实例化，可以通过继承解析器类来实现，改变实际的查询语法需要深入地了解 JavaCC 解析器生成器。为了实现更好的查询分析功能，可以分离一个查询的语法和语义。例如：'a AND b'与'+a +b'和'AND(a,b)' 是同一个查询的不同的语法。区分不同查询组件的语义，例如，是否符号化/原型化/正规化不同的词，并且如何实现，或者对于词用哪个查询对象来创建。需要能够尽可能快地用新的语法写一个解析器，重用底层的语义。

如果把查询看成是一个简单的语言，则可以用巴科斯范式 (BNF) 来定义查询语言的文法。

```

Query ::= ( Clause ) *
Clause ::= ["+", "-"] [<TERM> ":" ] ( <TERM> | "(" Query ")" )
  
```

这个文法中的每个规则都是一个产生式。每个产生式由左边的一个符号和右边的多个符号组成，用右边的多个符号来描述左边的一个符号。符号有终结符和非终结符两种。这里的终结符是 "+" 和 "-" 等，非终结符是 Clause 和 Query。例如，对于查询表达式



“site:lietu.com”，可以表示成<TERM>":"<TERM>的形式，最终归约成一个有效的 Query。

JavaCC 把非终结符对应成 Java 中的方法，例如非终结符 Clause 对应 Clause 方法。JavaCC 的 .jj 文件文法和标准巴科斯范式的区别是：在 JavaCC 中，可以在每步推导的过程中执行一些相关的 Java 语句，在编译原理中，把这些语句叫做动作。例如 StandardSyntaxParser.jj 中对 Clause 的定义：

```
QueryNode Clause(CharSequence field) : {
    QueryNode q; //定义局域变量
    Token fieldToken=null, boost=null;
    boolean group = false;
}
{
    [
        LOOKAHEAD(2) //向前看两个 Token
        (
            fieldToken=<TERM> <COLON> {field=EscapeQuerySyntaxImpl.discard
            EscapeChar(fieldToken.image);} //修改列名
        )
    ]

    (
        q=Term(field)
        | <LPAREN> q=Query(field) <RPAREN> (<CARAT> boost=<NUMBER>)?
        {group=true;} //如果碰到 ^ 字符则对查询加权
    )

    {
        if (boost != null) {
            float f = Float.valueOf(boost.image).floatValue();
            q = new BoostQueryNode(q, f);
        }
        if (group) { q = new GroupQueryNode(q); }
        return q; //返回生成的 QueryNode
    }
}
```

STATIC 是一个布尔选项，默认值是真。如果是真，在生成出的解析器和 token 管理器中，所有的方法和类变量都声明成静态的。这样仅允许一个解析对象存在，但是查询分析器应该有很多个，所以这个值应该设成假。

JavaCC 根据 StandardSyntaxParser.jj 转换出源代码：

```
Token.java
StandardSyntaxParserConstants.java
StandardSyntaxParser.java
...
```

`StandardSyntaxParser.jj` 中定义的 `parse` 方法定义了对用户查询串的词法分析功能，并完成初步的语法分析。

```
public QueryNode parse(CharSequence query, CharSequence field)
```

`QueryNode` 对象包含了分析出来的语法树。

查询解析器有三层，而它的核心是 `QueryNodeTree`。是一棵最初表示原始查询的语法树。例如，'a AND b' 的 `QueryNodeTree` 如图 6-2 所示。

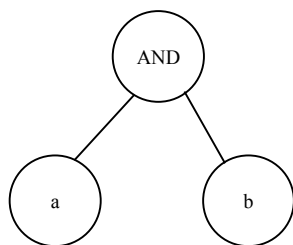


图 6-2 `QueryNodeTree`

查询解析器的三层具体说明如下。

- **QueryParser:** 最上层是解析层，简单地把查询字符串转换成一个 `QueryNodeTree`。当前使用 `Javacc` 实现这一层。
- **QueryNodeProcessor:** 查询节点处理器做了大部分工作。它实际上是一个可配置的处理器链。每一个处理器可以遍历树和修改节点甚至修改树的结构。这样就有可能在查询执行以前做查询优化或者把字符串分隔成词。
- **QueryBuilder:** 第三层也是一个构造器的可配置链，把 `QueryNodeTree` 转换成 Lucene 的 `Query` 对象。

新的 `QueryParser` 主要实现代码在 `contrib` 路径中的 `queryparser` 子路径下。

- `org.apache.lucene.queryParser.core:` 包含 query parser API 类，需要通过 query parser 实现扩展。

- org.apache.lucene.queryParser.standard: 包含使用新的 query parser API 的 query parser 实现。

使用 StandardQueryParser 的代码如下所示：

```
StandardQueryParser qpHelper = new StandardQueryParser();
StandardQueryConfigHandler config = qpHelper.getQueryConfigHandler();
config.setAllowLeadingWildcard(true);
config.setAnalyzer(new WhitespaceAnalyzer());
Query query = qpHelper.parse("apache AND lucene", "defaultField");
```

标准 QueryParser 中的实现代码展示了对这 3 个阶段的调用。

```
QueryNode queryTree = this.syntaxParser.parse(query, getField());
queryTree = this.processorPipeline.process(queryTree);
return (Query) this.builder.build(queryTree);
```

对 “+DisNey WOrld” 的解析过程如图 6-3 所示。

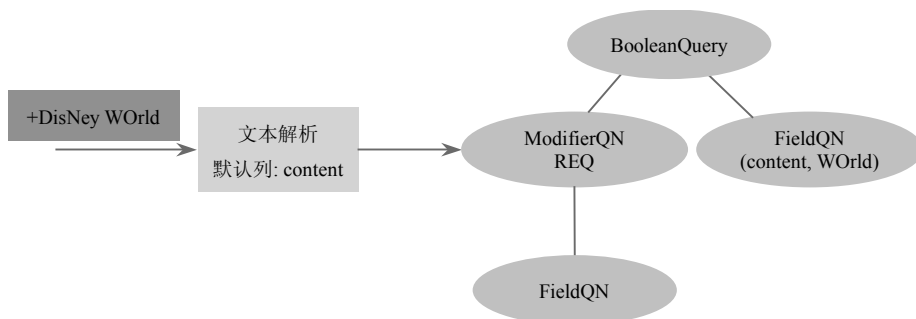


图 6-3 解析查询

需要让 QueryParser 更好地支持中文，例如查询语法支持全角空格，可以把全角空格当半角空格用。

### 6.1.3 查询原理

查询两个词 “NBA” 和 “AND 视频”，则对 “NBA” 这个词对应的文档编号列表和 “视频” 这个词对应的文档编号列表做交集（Intersection）运算后返回，示例代码如下所示：

```
sort(a); // 对数组 a 排序
sort(b); // 对数组 b 排序
int aindex = 0;
```

```
int bindex = 0;
while (aindex < a.length && bindex < b.length) {
    if (a[aindex] == b[bindex]) {
        System.out.println(a[aindex] + " is in both a and b.");
        aindex++;
        bindex++;
    }
    else if (a[index] < b[bindex]) {
        aindex++;
    }
    else {
        bindex++;
    }
}
```

Lucene 中的 ConjunctionScorer 类包含了类似的实现。

6.1.4 分析文本

如何表示文档和查询串中的文本？也就是说，应该索引什么样的字符串，按什么样的字符串查询。查询和索引英文的例子如表 6-2 所示。

表 6-2 文档和查询串中的文本

文 本	查 询
... Official Michael Jackson website ...	michael jackson
小写和大写	
... Michael Jackson's new video ...	michael jackson
切分问题	
... Fender Music, the guitar company ...	Fender guitars
词干化/原型化	
... Microsoft WindowsXP ...	windows xp
词边界	
... the cat is on the table ...	cat table
停用词	

全文索引是按词组织的，词是怎么来的呢？Analyzer 类分出来的。分析文本的工作交给了 Analyzer 类，这也是它唯一的工作。Lucene 把索引中的单词叫做 Token。Analyzer 把接收的字符串流解析成单词序列，也就是 Token 流，如图 6-4 所示。

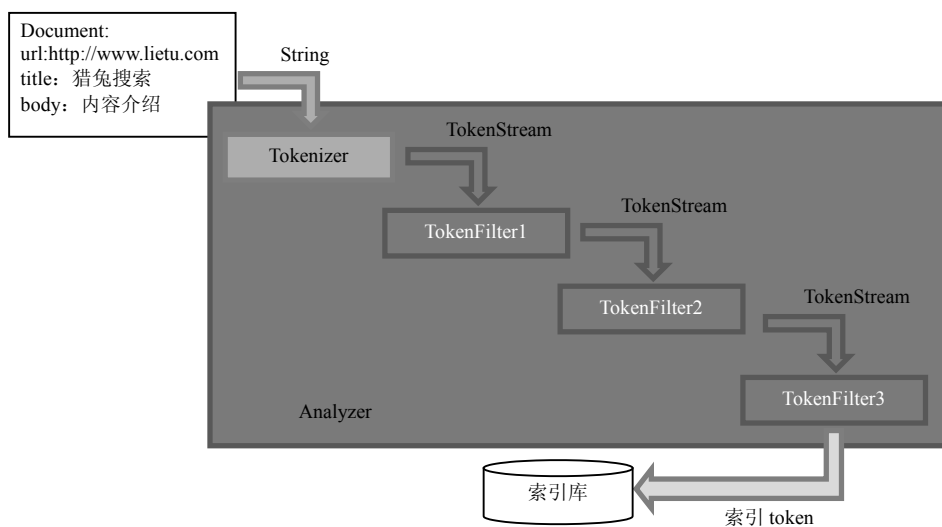


图 6-4 分析文档

Lucene 把查询串中的单词叫做 Term，如图 6-5 所示。

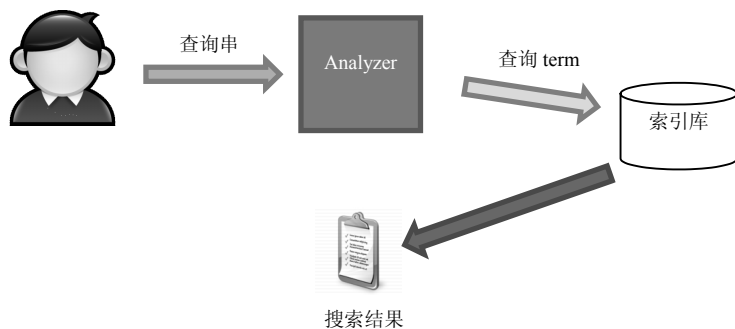


图 6-5 分析查询串

Lucene 在两个地方使用 Analyzer：索引文档的时候和按关键词搜索的时候。索引文档的时候 Analyzer 分析出的词成为倒排索引中的词。

一个好的 Analyzer 把文本转换成更适合检索的形式。例如搜索用户一般不关心单词是大写还是小写的，也就是说查找是大小写无关的。Analyzer 往往会做的一个简单的转换是小写化单词。

例如，小写化的 Analyzer 会把文档中的 CAT 转化成倒排索引中的 cat。也就是说 cat 会关联这个文档。

源文本是:

```
The Cat in the Hat.
```

分析器的输出是:

```
[the] [cat] [in] [the] [hat.]
```

注意这里用括号包装每一个 Token。

索引和搜索使用的 `Analyzer` 必须能够切出同样的 `Token`。如果索引的时候不使用 `LowerCaseFilter`，而搜索的时候使用 `LowerCaseFilter`，这样就会导致很多词搜索不出来。因为这时候用 `cat` 找不到 `CAT`。

往往对不同语言的文档和查询使用不同的 `Analyzer`。例如，`ClassicAnalyzer` 专门用来处理英文。`org.apache.lucene.analysis` 包含很多常用的 `Analyzer`。其中 `TokenStream` 类用来进行基本的分词工作。`Analyzer` 类是 `TokenStream` 的外围包装类，负责整个解析工作。有人把文本解析比喻成人体的消化过程，输入食物，分解出有用的氨基酸和葡萄糖等。`Analyzer` 类接收的是整段的文本，解析出有意义的词语。

`TokenStream` 是生产者，产生 `Token`。生成词索引的程序是消费者，调用 `TokenStream` 的 `incrementToken()` 方法得到一个 `Token`。

`TokenStream.incrementToken()` 方法修改内部状态，然后通过相关的 `Attribute` 得到词本身，以及它所在的位置信息等。`incrementToken()` 方法返回一个布尔值，如果返回 `true` 则表示后面还有 `Token` 可以取，如果返回 `false` 则表示后面没有更多的 `Token` 了。

```
String text = "The door has been opened for us.";

StandardAnalyzer analyzer = new StandardAnalyzer(Version.LUCENE_30);
TokenStream stream = analyzer.tokenStream("", new StringReader(text));

//增加 Token 表示的字符串属性
CharTermAttribute term = stream.addAttribute(CharTermAttribute.class);

while(stream.incrementToken()) {
    System.out.println(term.term()); //逐个单词输出
}
```





对于一个 `TokenStream` 的实例来说，每个 `Attribute` 的实现类都只有一个，这个 `TokenStream` 返回的所有 `Token` 都重用这个 `Attribute` 实现类。例如上面的 `CharTermAttribute` 实现类 `term` 只有一个。

```
CharTermAttribute termAtt = tokenStream.addAttribute(CharTermAttribute.class); //词
OffsetAttribute offsetAtt = tokenStream.addAttribute(OffsetAttribute.class); //用于高亮显示
tokenStream.addAttribute(PositionIncrementAttribute.class);
//用于处理同义词
```

`TokenStream.reset()` 不是在内部调用的，是在外部重复使用 `TokenStream` 的时候调用的。例如：

```
WhitespaceTokenizer tokenizer = new WhitespaceTokenizer(reader);
//消耗 Token
reader.reset();
tokenizer.reset(reader);
//可以再次消耗
```

自定义的 `Tokenizer` 往往需要重写 `reset()` 方法。

`TokenStream` API 的流程如下：

- 初始化 `TokenStream`，`TokenStream` 增加属性到 `AttributeSource`。
- 消费者调用 `reset()`。
- 消费者从 `TokenStream` 找属性，并把它想要得到的值存在本地变量中。
- 消费者调用 `incrementToken()` 直到它返回 `false`，每次调用后，消费属性值。
- 消费者调用 `end()`，以便执行流结束操作。

当结束使用 `TokenStream` 后，消费者调用 `close()` 来释放资源。

`TokenStream` 相关类的层次结构图如下：

```
* TokenStream

    * Tokenizer
        * KeywordTokenizer
        * CharTokenizer
        * WhitespaceTokenizer
```

```

        * LetterTokenizer
        * LowerCaseTokenizer
    * StandardTokenizer

    * TokenFilter
        * LowerCaseFilter
        * StopFilter
        * StandardFilter
        * PorterStemFilter
        * LengthFilter
        * ISOLatin1AccentFilter

```

一般在 `Tokenizer` 的子类实际执行词语的切分。需要设置的值有：和词相关的属性 `termAtt`、和位置相关的属性 `offsetAtt`。在搜索结果中高亮显示查询词时，需要用到和位置相关的属性。但是在切分用户查询词时，一般不需要和位置相关的属性。此外还有声明词类型的属性 `TypeAttribute`。`Tokenizer` 的子类需要重写 `incrementToken` 方法。通过 `incrementToken` 方法遍历 `Tokenizer` 分析出的词，当还有词可以获取时，返回 `true`，已经遍历到结尾时，返回 `false`。

基于属性的方法把无用的词特征和想要的词特征分隔开。每个 `TokenStream` 在构造时增加它想要的属性。在 `TokenStream` 的整个生命周期中都保留一个属性的引用。这样在获取所有和 `TokenStream` 实例相关的属性时，可以保证属性的类型安全。例如，在 `CnTokenStream` 中增加词属性。

```

protected CnTokenStream(TokenStream input) {
    super(input);
    termAtt = addAttribute(CharTermAttribute.class);
}

```

在 `TokenStream.incrementToken()`方法中，一个 `token` 流仅仅操作在构造方法中声明过的属性。例如，如果只要分词，则只需要 `CharTermAttribute`。其他的属性，例如 `PositionIncrementAttribute` 或者 `PayloadAttribute` 都被这个 `TokenStream` 忽略掉了，因为这时不需要其他的属性。

“wi-fi”这个词用 `WordDelimiterFilter` 创建出如下三个 `Token`：

```

wi(posinc=1), fi(posinc=1), wifi(posinc=0)

```



这种做法，'wifi'简单地堆放在'fi'上是一个损失。PositionLengthAttribute 修正了这个错误，它允许一个词声明它的跨度，因此不会丢失任何信息。

Lucene 的 TokenStream 类产生文档字段中要索引的 Token 序列。API 是一个迭代器：调用 incrementToken 推进到下一个 Token，然后通过查询特定的属性来获得当前 Token 的细节。

例如，CharTermAttribute 持有 Token 的文本。为了能够突出显示这个词，OffsetAttribute 中有原始字符串中对应此 Token 的字符开始和结束偏移量。

TokenStream 实际是一个链。从一个 Tokenizer 开始，把输入字符串初步分解成 Token 序列，然后再用任意数量的 TokenFilter 修改或者插入已有的 Token 序列。TokenStream 是一个抽象类，Tokenizer 和 TokenFilter 是它的两个子类。Tokenizer 用 StringReader 构造，而 TokenFilter 则用另外一个 TokenStream 构造。

也可以使用 CharFilter 在分出 Token 前预处理字符，例如去掉 HTML 标记，或者根据正则表达式替换字符，同时保留合适的偏移量到原来的输入字符串。Analyzer 是根据需要创建 TokenStream 的工厂类。

Lucene 包含支持 34 种语言的 Tokenizer 和 TokenFilter。包括英文、中文、韩文、德文、法文、日文、阿拉伯文、俄文、西班牙文、葡萄牙文等。

StandardAnalyzer 可以用来分析 Unicode 文本。ClassicAnalyzer 专门用来处理英文，所以速度比 StandardAnalyzer 快。StandardAnalyzer 返回一个 TokenStream 的实例。

标记化一个简单的例子：fast wi fi network is down. 假设保留停用词。当看成一个图时，Token 看起来如图 6-6 所示。

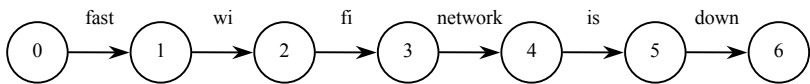


图 6-6 TokenStream 词图

每个节点是一个位置，而每个边是一个 Token。TokenStream 列举一个有向无环图，每次一个弧。TokenStream 本质上是根据要形成倒排索引的文本生成一个词图。

接下来，增加 SynonymFilter 到分析链，应用下面这些同义词：

```

fast → speedy
wi fi → wifi
wi fi network → hotspot

```

产生如图 6-7 所示的 TokenStream 词图。

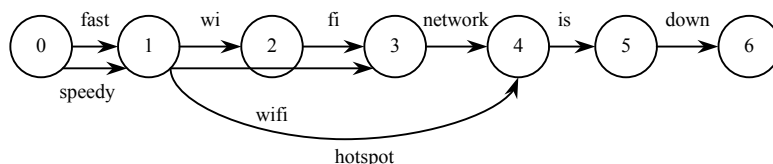


图 6-7 增加同义词后的 TokenStream 词图

PositionIncrementAttribute 告诉我们当前 Token 从边的开始位置前进了多少个位置。PositionIncrementAttribute 的值默认是 1。例如，对于 Token[fast] 来说，PositionIncrementAttribute 的值就是 1。

PositionLengthAttribute 说明边向前到达多少位置，也就是前进多少个节点。例如，Token[hotspot]的 PositionLengthAttribute 值是 3。

OffsetAttribute 中虽然保存了词的开始和结束位置的信息，但只是用于高亮显示，没有用于查询过程。

除了 SynonymFilter，还有几个其他的分析组件产生词图。例如 Kuromoji 的 JapaneseTokenizer 对于复合 Token 输出分解后的词。例如，ショッピングセンター（购物中心）这样的 Token 也会有一个可选的路径ショッピング（购物）跟着センター（中心）。很多中文词是复合词，因此也需要用词图来描述词之间的关系。日文分词包 kuromoji 用到了 PositionLengthAttribute：

```

public final class JapaneseTokenizer extends Tokenizer {
    private final PositionLengthAttribute posLengthAtt =
        addAttribute(PositionLengthAttribute.class);

    @Override
    public boolean incrementToken() throws IOException {
        //...
        posLengthAtt.setPositionLength(token.getPositionLength());
        //...
    }
}

```



当 ShingleFilter 和 CommonGramsFilter 合并两个输入 Token 的时候, 设置位置长度为 2, 也就是把 PositionLengthAttribute 设置成 2。

```
public final class ShingleFilter extends TokenFilter {
    private final PositionLengthAttribute posLenAtt =
        addAttribute(PositionLengthAttribute.class);

    public final boolean incrementToken() throws IOException {
        //...
        posLenAtt.setPositionLength(builtGramSize);
        //...
    }
}
```

对于 CommonGramsFilter 来说, 输入 "the quick brown fox", 输出 ["the", "the-quick", "brown", "fox"]。Token[the-quick]的 PositionIncrementAttribute 的值就是 0, 而 PositionLengthAttribute 的值是 2。

此外, WordDelimiterFilter、DictionaryCompoundWordTokenFilter、HyphenationCompoundWordTokenFilter、NgramTokenFilter、EdgeNGramTokenFilter 等还需要增加 PositionLengthAttribute 的值。

但是, 索引目前忽略 PositionLengthAttribute, 它只注意到 PositionIncrementAttribute。需要修改 Codec API, 然后让位置相关的查询用到这个值。

查询分析器也忽略了位置长度。如果增加对 PositionLengthAttribute 的支持, 就可以在查询时运行图式分析器, 用词图做查询扩展, 更容易得到正确的结果。

### 6.1.5 使用 Filter 筛选搜索结果

可以定义 Filter 类来过滤查询结果。也可以缓存和重用 Filter。如下条件可用 Filter 来实现:

- 根据不同的安全权限显示搜索结果;
- 仅查看上个月的数据;
- 在某个类别中查找。

下面定义一个 BestDriversFilter, 把搜索结果限定到 score 是 5 的司机。

```

public class BestDriversFilter extends Filter{
    @Override
    public DocIdSet getDocIdSet(IndexReader reader) throws IOException{
        OpenBitSet bitSet = new OpenBitSet( reader.maxDoc() );
        //查询出 score 是 5 的文档
        TermDocs termDocs = reader.termDocs( new Term( "score", "5" ) );
        while ( termDocs.next() ) {
            bitSet.set( termDocs.doc() );//把符合条件的文档对应的位置为 1
        }
        return bitSet;
    }
}

```

在查询中使用这个 Filter:

```

Filter bestDriversFilter = new BestDriversFilter();
//query 不变, 增加 bestDriversFilter
ScoreDoc[] hits = isearcher.search(query, bestDriversFilter, 1000).scoreDocs;
//因为不是每个司机都能得 5 分, 所以返回的结果可能比以前少了

```

### 6.1.6 遍历索引库

经常需要统计索引库中哪些词出现的频率最高。例如, 需要统计旅游活动索引库中的热门目的地。可以先对目的地列做索引, 索引列不分词, 然后取得该列中最常出现的几个词也就是热门目的地。实现方法是: 用 `TermEnum` 遍历索引库中所有的词, 取出每个词的文档频率, 然后使用优先队列找出频率最高的几个词。

```

public class TermInfo {
    public Term term;    //索引库中的词
    public int docFreq;  //文档频率, 也就是这个词在多少文档中出现过

    public TermInfo(Term t, int df) {
        this.term = t;
        this.docFreq = df;
    }
}

public static TermInfo[] getHighFreqTerms(IndexReader reader,
                                           int numTerms, String field){
    //实例化一个 TermInfo 的队列
    TermInfoQueue tiq = new TermInfoQueue(numTerms);
    TermEnum terms = reader.terms(); //读取索引文件里所有的 Term
    int minFreq = 0; //队列最后一个 Term 的频率即当前最小频率值
    while (terms.next()) { //取出一个 Term 对象出来
        String field = terms.term().field();
    }
}

```



```
if (fields != null && fields.length > 0) {
    boolean skip = true; //跳过标识
    for (int i = 0; i < fields.length; i++) {
        //当前 Field 属于 fields 数组中的某一个则处理对应的 Term
        if (field.equals(fields[i])) {
            skip = false;
            break;
        }
    }
    if (skip) continue;
}
//当前 Term 的内容是过滤词，则直接跳过
if (junkWords != null && junkWords.get(terms.term().text()) !=
    null) continue;

//获取最高频率的 Term。基本方法是：
//队列底层是最大频率 Term，顶层是最小频率 Term，
//当插入一个元素后超出初始化队列大小则取出最上面的那个元素，
//重新设置最小频率值 minFreq
if (terms.docFreq() > minFreq) {
//当前 Term 的频率大于最小频率则插入队列
    tiq.insertWithOverflow(new TermInfo(terms.term(), terms.
        docFreq()));
    if (tiq.size() >= numTerms) {
        //当队列中的元素个数大于 numTerms
        tiq.pop(); // 取出最小频率的元素即最上面的一个元素
        minFreq = ((TermInfo)tiq.top()).docFreq;
        //重新设置最小频率
    }
}
}
//取出队列元素，最终存放在数组中元素的词频率按从大到小排列
TermInfo[] res = new TermInfo[tiq.size()];
for (int i = 0; i < res.length; i++) {
    res[res.length - i - 1] = (TermInfo)tiq.pop();
}
return res;
}
```

### 6.1.7 索引数值列

一个索引库类似一个数据库的表结构，但是在 Lucene 2.9 以前的版本中只能存储字符串，如果是日期或者数字，需要用专门的方法转换成字符串后再索引。新的版本可以直接存储数字：

```
document.add(new NumericField(name).setIntValue(value));
```

高级搜索中可能会用到范围查询，例如查询价格区间。

Lucene 2.9 以前版本实现的区间查询性能上有问题。`RangeQuery` 采用扩展成 `TermQuery` 来实现，如果查询区间范围太大，`RangeQuery` 会导致 `TooManyClausesException`。为了避免产生这个异常，`ConstantScoreRangeQuery` 没有采用扩展成布尔查询的方式实现，而是采用 `Filter` 来实现，但是当索引很大的时候，查询速度会变慢。

在 Lucene 2.9 以后的版本中，用 `Trie` 结构索引日期和数字等类型。例如，把 521 这个整数索引成“百位是 5、十位是 52、个位是 521”。这样重复索引的好处是可以用最低的精度搜索匹配区域的中心地带，用较高的精度匹配边界。这样减少了要搜索的 `Term` 数量。例如，`TrieRange:[423 TO 642]` 分解为 5 个子条件来执行：

hundreds:5 OR tens:[43 TO 49] OR ones:[423 TO 429] OR tens:[60 TO 63] OR ones:[640 TO 642]

`TrieRange` 的工作原理如图 6-8 所示。

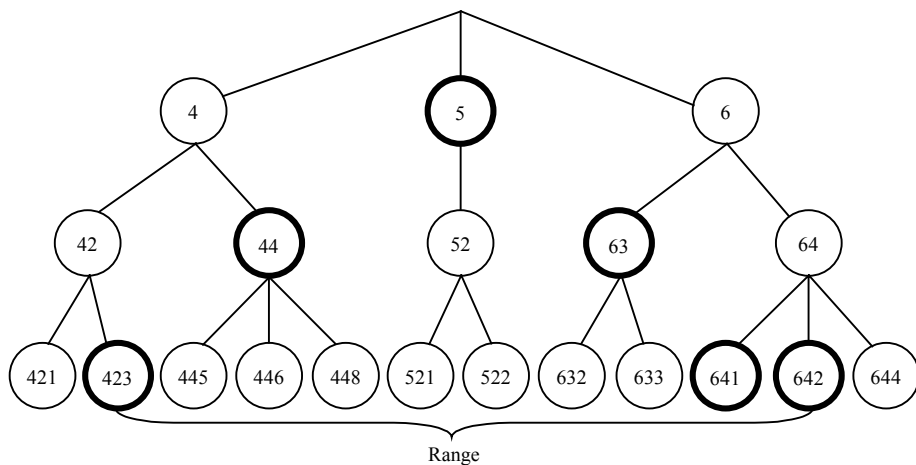


图 6-8 `TrieRange` 工作原理

可以用 `NumericField` 来实现 `Trie` 结构索引数字。这样做的好处是能够更有效地按区间查找数字和排序。下面是增加一个整数列到索引的例子：

```
document.add(new NumericField(name).setIntValue(value));
```





为了优化索引性能，可以重用 `NumericField` 和 `Document` 实例：

```
NumericField field = new NumericField(name);
Document document = new Document();
document.add(field);

for(all documents) {
    ...
    field.setIntValue(value)
    writer.addDocument(document);
    ...
}
```

然后可以使用 `NumericRangeQuery` 来查询这样的数字列。例如：

```
Query q = NumericRangeQuery.newFloatRange("weight", //列名称
    new Float(0.3f), //最小值从它开始
    new Float(0.10f), //最大值到它结束
    true, //是否包含最小值
    true); //是否包含最大值
```

如果要索引日期列，且是精确到天的搜索，只需要把日期类型的值用 `DateTools` 转换成“yyyyMMdd”格式的字符串，然后再转换成整数来索引就可以了。比如：

```
DateTime d=DateTime.Now;
int n= Integer.valueOf(d.ToString("yyyyMMdd"));
doc.Add(new NumericField("num", Field.Store.YES, true).SetIntValue(if));
```

如果精确到秒，需要先用 `Date.getTime()` 把日期转换成 `long` 类型。索引过程如下所示：

```
NumericField documentDateTimeField = new NumericField("Pub_Date",
    1,Field.Store.NO,
    true);

Document document = new Document();
document.add(documentDateTimeField);

while(hasDoc) {
    documentDateTimeField.setLongValue(scoreDetails.getDocumentDate()
        .getTime());
    writer.addDocument(document);
}
```

查询过程如下所示：

```

Long begin = esq.getBeginDate().getTime(); //开始时间
Long end = esq.getEndDate().getTime(); //结束时间

NumericRangeQuery rangeQuery = NumericRangeQuery.newLongRange("Pub_
Date",
                                1, begin, end,
                                esq.isBeginDateInclusive(),
                                esq.isEndDateInclusive());

```

如果精确到天，可以用 `DateTools` 把日期转换成 `int` 类型。搜索过程如下所示：

```

int int_Date_Begin = Integer.valueOf(DateTools.dateToString(date1,
Resolution.DAY));
int int_Date_End = Integer.valueOf(DateTools.dateToString(date2,
Resolution.DAY));

SortField sortField = new SortField("Pub_Date",SortField.INT,false);
Sort sort = new Sort(sortField);

//检索
NumericRangeQuery rangeQuery = NumericRangeQuery.newIntRange("Pub_Date",
                                int_Date_Begin,int_Date_End,
                                true, true);
ScoreDoc[] hits = searcher.search(rangeQuery, null, 1000, sort).scoreDocs;

```

可能需要索引数值列并按数值排序：

```

NumericField priceField = new NumericField("price");
Document document = new Document();
document.add(priceField);

for(all documents) {
    ...
    priceField.setIntValue(value);
    writer.addDocument(document);
    ...
}

```

按价格升序排列：

```

Sort sort= new Sort(new SortField("price",SortField.INT,false));
ScoreDoc[] hits = searcher.search(query,null,1000,sort).scoreDocs;

```



## 6.2 Lucene 中的压缩算法

倒排索引的大小可能和文档内容本身差不多大。当文档数量多时，倒排索引也会很大。为了节省存储空间，可以采用压缩格式存储倒排索引。从一个硬盘读入索引数据时，采用压缩存储后，能降低磁头需要移动的距离，从而提高性能。另外，为了在搜索系统内部快速地传输文档编号数组，可以压缩文档编号。

因为信息存在冗余，所以可以压缩。例如在生活中登记信息时，如果发现当前这行信息和上面一行信息相同时，为了少写字，可以采用压缩写法“同上”。这样的压缩原理叫做预测编码（Predictive Encoding），可以对前后相似的内容进行压缩。

压缩算法存在两个过程：编码过程和解码过程。编码过程也就是压缩过程，而解码过程也就是解压缩过程。编码过程可以时间稍长，而解码过程则需要速度快。这有点类似 ADSL 优化上网速度的机制：用户往往下载文件的时候多，而上传文件的时候少。所以 ADSL 设计成下载速度快，而上传速度慢。因为在索引数据阶段执行编码过程，而在搜索阶段执行解码过程，所以索引数据的速度可以稍慢，但是搜索速度不能慢。因为索引一般只需要在后台完成一次，而搜索则需要经常调用。

### 6.2.1 变长压缩

全文索引中的文档编号和词频都是正整数，所以 Lucene 的内部实现需要考虑压缩存储正整数的问题。压缩的原理是出现次数较多的数用较短的编码表示。这样一来，变短的数相对于变长的数更多，文件的总长度就会减少。Lucene 中的文档编号是一个正整数。在倒排索引中，小的数字出现概率大，如图 6-9 所示，说明较小的数值用较短的编码可以取得不错的压缩效果。

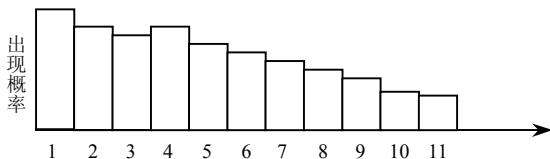


图 6-9 倒排索引中数值的出现频率

Lucene 采用了变长压缩方法（Variable byte encoding）。变长压缩算法的原理是：较小的数使用较短的编码，较大的数使用较长的编码。和压缩相关的类有用于压缩单个整数的 `VInt` 类和压缩排好序的整数数组的 `SortedVIntList` 类。为了实现更好的

的压缩，在有些地方可以使用 `PForDelta` 来代替 `VInt`。

VInt 是一个变长的正整数表示格式，是一种整数的压缩格式表示方法。每字节分成两部分，最高位和剩下的低 7 位。最高位表明是否有更多的字节在后面，0 表示这个字节是尾字节，1 表示还有后续字节。低 7 位表示数值。按如下的规则编码正整数  $x$ 。

- if ( $x < 128$ )，则使用一个字节（最高位置 0，低 7 位表示数值）；
- if ( $x < 128 \times 128$ )，则使用两个字节（第一个字节最高位置 1，低 7 位表示低位数值，第二个字节最高位置 0，低 7 位表示高位数值）。
- if ( $x < 128^3$ )，则使用 3 个字节，依次类推。

每字节的低 7 位表明整数的值，可以把 VInt 看成是 128 进制的表示方法，低位优先，也就是说随着数值的增大，向后面的字节进位，从 VInt 值表示示例表 6-3 可以看出。

表 6-3 VInt 编码示例

值	二进制编码	16 进制编码
0	00000000	00
1	00000001	01
2	00000010	02
127	01111111	7F
128	10000000 00000001	00 01
129	10000001 00000001	01 01
130	10000010 00000001	02 01
16383	11111111 01111111	7F 7F
16384	10000000 10000000 00000001	00 00 01
16385	10000001 10000000 00000001	01 00 01

从 0 到 127 的值可以存储在一个字节中，从 128 到  $128 \times 128 = 16383$  的值可以存储在 2 个字节中。这里  $16,383 = 127 \times 128 + 127$ ，二进制表示方式是 11111111 01111111。可以认为 Vint 能表示的整数值范围没有上限。因为小的数值更经常出现，相对于 int 的 4 个字节的表示方法而言，VInt 表示方法对小的数值使用的字节数更少。所以 VInt 提供了正整数的压缩表示，而且解码效率较高。

org.apache.lucene.store.IndexOutput.writeVInt(int i)方法包含了正整数编码成 VInt 的实现：

```
while ((i & ~0x7F) != 0) {
    writeByte((byte)((i & 0x7f) | 0x80)); //先写入低位字节
    i >>= 7; //右移 7 位
}
writeByte((byte)i); //写入高位字节
```

`org.apache.lucene.store.IndexInput.readVInt()`方法包含了 VInt 解码方法的实现：

```
byte b = readByte(); //读入一个字节
int i = b & 0x7F; //取低 7 位的值
//每个高位的字节多乘个 2 的 7 次方，也就是 128
for (int shift = 7; (b & 0x80) != 0; shift += 7) {
    b = readByte();
    i |= (b & 0x7F) << shift; //当前字节表示的位乘 2 的 shift 次方
}
return i; //返回最终结果 i
```

## 6.2.2 PForDelta

VInt 对每个整数都选择不同的位数来编码，因为有判断分支，导致解码速度慢。和 VInt 按单个整数的压缩算法不同，PForDelta 压缩算法按批压缩整数，每批可以选取连续的 128 个整数。把 128 个整数中 90% 较小的数用统一长度的短编码表示。把剩下 10% 的整数看成是异常的大数，异常数用普通的 4 个字节的 int 型来表示。一批数的空间有 128 个  $b$  位槽空间。使用没有用到的  $b$  位槽空间构造链接表，一个异常数的  $b$  位槽空间存储碰到下一个异常数的偏移量，异常数的原始值追加存储在后面。VInt 算法对每个整数的编码长度都不一样，而 PForDelta 压缩算法对于一批整数只有两个编码长度，分别是  $b$  和 4。

例如，可以通过直方图来统计用多少位就可以表示这批数据中的 90%，如果统计出来是  $b$  位，也就是说，有 90% 的数小于  $2^b$ 。

假设在 128 个整数中，其中 90% 的数是小于 32 的，另外 10% 是大于 32 的，那么就把这 128 个整数以 32 为界，把小于 32 的作为一种处理，大于 32 的作为异常情况进行处理。32 是 2 的 5 次方，因此  $b=5$ 。整个压缩空间由两部分组成：分配  $128*5$  位给 90% 的正常数的空间，加上分配给剩下 10% 的异常数的空间。例如， $b=5$  整数序列是：23, 41, 8, 12, 30, 68, 18, 45, 21, 9... 压缩存储成如图 6-10 所示的形式。

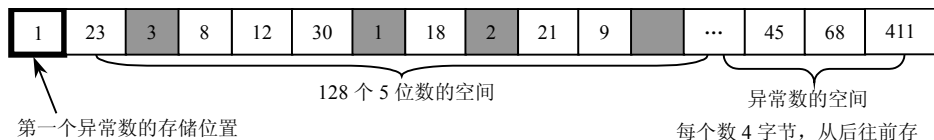


图 6-10 PForDelta 压缩空间示例

为了处理方便，如果有连续 32 个相邻的数是小于 32 的，那么把第 32 个数强制作异常情况进行处理。

开源项目 Kamikaze (<http://sna-projects.com/kamikaze/>) 中提供了 PforDelta 的 Lucene 实现版本。Kamikaze 来源于 LinkedIn 网站，是一个 Java 工具包，封装了文档 id 号集合的实现，它实现了排序的整数段的 PForDelta 压缩算法。Kamikaze 也为 Lucene 提供了倒排索引列表的压缩。PForDelta 压缩算法实现在 `com.kamikaze.docidset.compression.P4DsetNoBase` 的 `compressAlt` 方法如下所示：

```
public long[] compressAlt(int[] input){
    //默认情况下，b=32，编码长度是 32
    int BATCH_MAX = 1 << (_b - 1);

    //分配数据空间大小，这里 BASE_MASK=32
    long[] compressedSet = new long[(((batchSize) * _b + HEADER_MASK
    + _exceptionCount * (BASE_MASK))>>>6)+1];

    //把 b 加入数据空间
    copyBits(compressedSet, _b, 0, BYTE_MASK);

    //Offset 是下一个存储值的位置偏移量，在这里 HEADER_MASK 的值是 8
    int offset = HEADER_MASK;
    int exceptionOffset = _exceptionOffset;
    int exceptionIndex = 0;

    //遍历需要压缩的数组
    for (int i = 0; i < _batchSize; i++) {
        //如果是正常数则放在前面的空间，否则拷贝到压缩空间最后的位置
        if (input[i] < BATCH_MAX) {
            copyBits(compressedSet, input[i] << 1, offset, _b);
        } else {
            //记录异常点的编号到 b 位槽空间，并且增加一个位标志
            copyBits(compressedSet, ((exceptionIndex << 1) | 0x1), offset, _b);

            //把异常数值放到数据空间的最后的位置
            copyBits(compressedSet, input[i], exceptionOffset, BASE_MASK);

            //增加数据空间最后位置的长度
            exceptionOffset += BASE_MASK;
            exceptionIndex++;
        }

        offset += _b;
    }
    return compressedSet;
}
```



PforDelta 的解码过程如下所示：

```
public int[] decompress(long[] compressedSet) {
    int[] op = new int[_batchSize];
    //reuse o/p
    op[0] = _base;

    //异常列表的偏移量
    int exceptionOffset = HEADER_MASK + _b * _batchSize;

    //扩展和修补数据
    for (int i = 1; i < _batchSize; i++) {
        int val = getBitSlice(compressedSet, i * _b + HEADER_MASK, _b);

        if ((val & 0x1) != 0) {
            //碰到一个异常
            op[i] = getBitSlice(compressedSet, exceptionOffset, BASE_MASK);
            exceptionOffset += BASE_MASK;
        } else {
            op[i] = val >>> 1;
        }
        op[i] += op[i - 1];
    }
    return op;
}
```

PForDelta 算法在实际的使用虽然压缩速度稍慢，但是有非常好的解压速度。因为压缩过程在索引阶段完成，解压过程在搜索阶段完成，所以在全文索引库实现中流行采用 PForDelta 压缩算法。解压速率快的原因是它符合现代计算机的流水线工艺——即解压过程中没有判断语句，不会打断 CPU 的流水线；同时每次 128 个数据可以缓存在 CPU 内。对比 VInt 算法，由于 VInt 中的 128 是一个较小的数，无论怎样安排判断语句的顺序，都会造成 CPU 流水线的预测失误，从而造成较大的性能损失。

### 6.2.3 前缀压缩

对于全文索引中的索引词也进行了压缩存储。索引词压缩算法采用了前缀压缩（Front Encoding）。因为索引词是排序后写入索引的，所以前后两个索引词的词形差别往往不大。前缀压缩算法省略存储相邻两个单词的共同前缀。每个词的存储格式是：

<相同前缀的字符长度,不同的字符长度,不同的字符>

例如，顺序存储如下三个词：term、termagancy、termagant。不用压缩算法的存储方式是：

```
<4,term> <10,termagancy> <9,termagant>
```

如果应用前缀压缩算法，实际存储的内容如下所示：

```
<4,term> <4,6, agancy> <8,1,t>
```

在 TermInfosWriter 类中实现的前缀压缩代码如下所示：

```
int start = 0;
final int limit = termBytesLength < lastTermBytesLength ? termBytesLength :
lastTermBytesLength;
while(start < limit) {
    if (termBytes[start] != lastTermBytes[start])
        break;
    start++;
}

final int length = termBytesLength - start;
output.writeVInt(start);           //写入相同前缀的长度
output.writeVInt(length);          //写入增量长度
output.writeBytes(termBytes, start, length); //写入增量字节
if (lastTermBytes.length < termBytesLength) {
    byte[] newArray = new byte[(int) (termBytesLength*1.5)];
    System.arraycopy(lastTermBytes, 0, newArray, 0, start);
    lastTermBytes = newArray;
}
System.arraycopy(termBytes, start, lastTermBytes, start, length);
lastTermBytesLength = termBytesLength;
```

在 TermVectorsReader 类中实现的解压缩代码如下所示：

```
start = tvf.readVInt(); //读入相同前缀的长度
deltaLength = tvf.readVInt(); //读入增量长度
totalLength = start + deltaLength;

final String term; //要解压缩的索引词

if (byteBuffer.length < totalLength) {
    byte[] newByteBuffer = new byte[(int) (1.5*totalLength)];
    System.arraycopy(byteBuffer, 0, newByteBuffer, 0, start);
    byteBuffer = newByteBuffer;
```





```

    }
    tvf.readBytes(byteBuffer, start, deltaLength);
    term = new String(byteBuffer, 0, totalLength, "UTF-8");

```

此外，字符串压缩算法还有 BurrowsWheeler 转换等。

## 6.2.4 差分编码

变长压缩算法对于较小的数字有较好的压缩比。差分编码（Differential Encoding）可以把数组中较大的数值用较小的数来表示，所以可以和变长压缩算法联合使用来实现压缩。差分编码中存储的是数组序列中前后两个数之间的差异。差分编码压缩的过程示例如下所示：

$$X_1, X_2, L \Rightarrow X_1, X_2 - X_1, L, X_n - X_{n-1}$$

差分编码解压缩的过程示例如下所示：

$$Y_1, Y_2, L, Y_n \Rightarrow Y_1, Y_2 + Y_1, L, Y_n + Y_{n-1}$$

例如，对于排好序的 DocId 序列，其

编码前是：345, 777, 11437, ...

编码后是：345, 432, 10660, ...

在压缩过程中，先取出一个已经从小到大排好的数组的第一个数字，通过 longToBytes 方法把此数值转换成一个 byte 数组，用 BufferedOutputStream 实例把它写入文件，然后通过一个循环，用一个变量记录数组中后一个数减前一个数的差，如果第二个数等于数组的第二个数减去数组的第一个数，再通过 writeVLong 方法将这个变量写入文件中。

在解压缩过程中，先用 DataInputStream 实例的 readLong 方法读出文件的第一个数字写入数组，再通过 readVLong 方法将该文件剩下的数字读取出来，同时把此数组剩下的数字都变成后一个数字和前一个数字的和，还原成原来的数据。

```

//把一个 long 型的数据变成二进制
public static byte[] longToBytes(long n) {
    byte[] buf=new byte[8]; //新建一个 byte 数组
    for(int i=buf.length-1;i>=0;i--){
        buf[i]=(byte) (n&0x00000000000000ff); //取低 8 位的值
    }
}

```

```

        n>>>=8; //右移 8 位
    }
    return buf;
}
//把一个 long 型的数据进行压缩
public static void writeVLong(long i,BufferedOutputStream dos) throws
IOException{
    while ((i & ~0x7F) != 0) {
        dos.write((byte)((i & 0x7f) | 0x80)); //写入低位字节
        i >>>= 7; //右移 7 位
    }
    dos.write((byte)i);
}
//把一个压缩后的 long 型的数据读取出来
private static long readVLong(DataInputStream dis) throws IOException{
    byte b = dis.readByte(); //读入一个字节
    int i = b & 0x7F; //取低 7 位的值
    //每个高位的字节多乘个 2 的 7 次方，也就是 128
    for (int shift = 7; (b & 0x80) != 0; shift += 7) {
        if(dis.available()!=0){
            b = dis.readByte();
            i |= (b & 0x7F) << shift; //当前字节表示的位乘 2 的 shift 次方
        }
    }
    return i; //返回最终结果 i
}
//把 long 型数组 simHashSet 写入 fileName 指定的文件中
private static int write(long[] simHashSet,String fileName) {
    BufferedOutputStream dos =new BufferedOutputStream(new File
    OutputStream(fileName));
    byte[] b = longToBytes(simHashSet[0]);
    //数组的第一个数字一个转换成二进制
    dos.write(b); //把它写到文件中
    for (int i = 1; i < simHashSet.length; i++) {
        long deta=simHashSet[i]-simHashSet[i-1];
        //数组中后一个数减前一个数的差
        writeVLong(deta, dos); // 把这个差值写入文件
    }
    dos.close();
    return simHashSet.length;
}
//从 fileName 指定的文件中把 long 型数组读出来
private static long[] read(int len,String fileName) {

```

```

    DataInputStream dis = new DataInputStream(new BufferedInputStream(
        new FileInputStream(fileName)));
    long[] simHashSet = new long[len];
    simHashSet[0] = dis.readLong(); //从文件读取第一个 long 型数字放入数组
    for (int i = 1; i < len; i++) {
        simHashSet[i] = readVLong(dis); //读取文件剩下的元素
        //将元素都变成数组后一个数和前一个数字的和
        simHashSet[i] = simHashSet[i] + simHashSet[i - 1];
    }
    dis.close();
    return simHashSet;
}

```



## 6.3 创建和维护索引库

可以用 Lucene 提供的 API 创建和更新索引。在生成索引过程中涉及的几个类关系如图 6-11 所示。

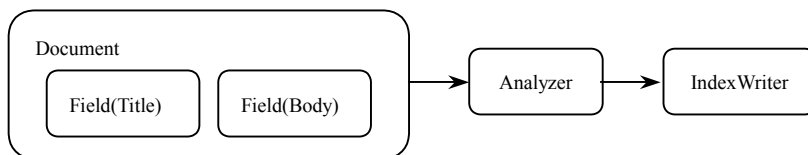


图 6-11 创建索引过程中用到的类

### 6.3.1 创建索引库

索引一般存放在硬盘中的一个路径中。可以通过 `IndexWriter` 来创建一个新的索引库。相关的参数有：索引路径和配置信息。

```

static Version matchVersion=Version.LUCENE_43;
Analyzer analyzer=new StandardAnalyzer();

IndexWriterConfig iwc = new IndexWriterConfig(matchVersion, analyzer);

Directory dir = FSDirectory.open(new File(indexPath));
IndexWriter index = new IndexWriter(dir, iwc); //指定索引路径和配置信息

```

如果有多个索引，一般不会放在同一个索引。除了在可以长期保存的物理路径中，索引库也可以仅在内存中。在测试分词或索引的时候，可能会用到内存索引。

```
Directory dir = new RAMDirectory(); //内存路径
IndexWriter index = new IndexWriter(dir, iwc);
```

### 6.3.2 向索引库中添加索引文档

往索引添加数据时涉及的几个类关系如图 6-12 所示。

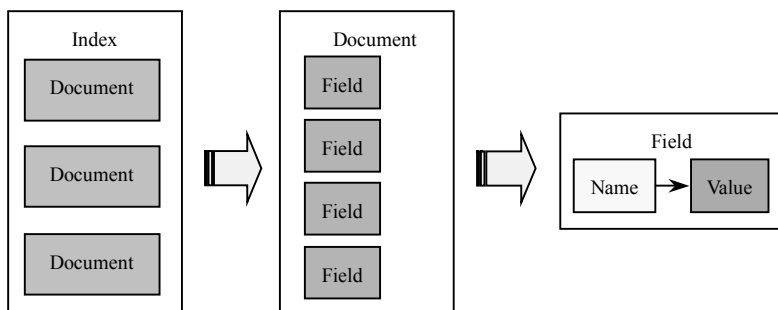


图 6-12 往索引中添加文档

下面这段程序向索引库中添加网页地址，标题和内容列：

```
Document doc = new Document();
Field f = new Field("url", news.URL ,
    Field.Store.YES, Field.Index.UN_TOKENIZED,
    Field.TermVector.NO);
doc.add(f);
f = new Field("title", news.title ,
    Field.Store.YES, Field.Index.TOKENIZED,
    Field.TermVector.WITH_POSITIONS_OFFSETS);
doc.add(f);
f = new Field("body", news.body.toString() ,
    Field.Store.YES, Field.Index.TOKENIZED,
    Field.TermVector.WITH_POSITIONS_OFFSETS);
doc.add(f);
index.addDocument(doc);
```

**【作者提示】**如果初次使用 Lucene，往索引中写入的每条记录最好都新建一个 Document 与之对应，也就是说 Document 对象不要重用，否则可能会出现意想不到的结果。

索引创建完成后可以用索引查看工具 Luke (<http://code.google.com/p/luke/>) 来查看索引内容并维护索引库。Luke 是一个可以执行的 jar 包，是用 Java 实现的 windows 程序。在 Windows 下可以双击 lukeall-1.0.1.jar，启动 Luke。选择菜单“File”→“Open Lucene index”，打开“data/index”文件夹（如图 6-13 所示），然后可以在窗口中看到索引创建的详细信息

（如图 6-14 所示）。

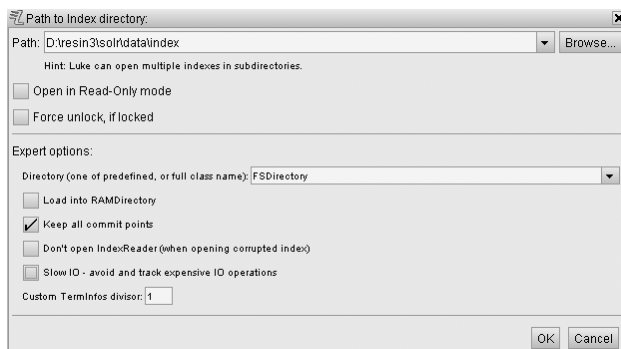


图 6-13 打开索引

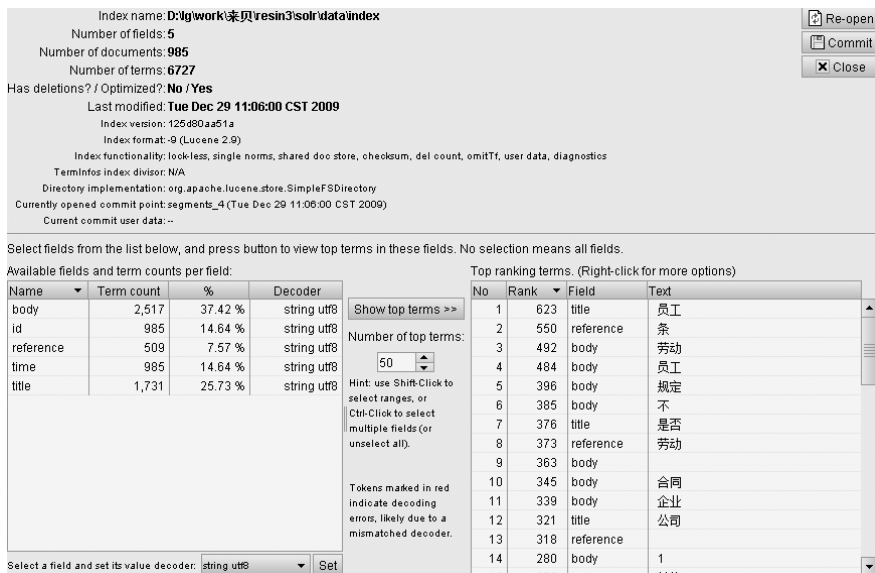


图 6-14 索引的概要信息

为了提高索引速度，可以重用 Field，而不是每次都创建新的。从 Lucene 2.3 开始，有新的 setValue 方法，可以改变一个 Field 的值。这样可以在增加许多 Document 的时候重用单个的 Field 实例，可以节省许多 GC 消耗的时间。

最后新建一个独立的 Document 实例，然后增加许多 Field 实例，并且增加每个文档到索引的时候都重用这些 Field。例如，有一个 idField 和 bodyField、nameField 等。当加入一个 Document 后，可以通过 idField.setValue(...)等直接改变 Field 值，然后再增加文档实例。下面是一个重用 Field 的例子：

```

Field idField = new Field("cid", null ,
    Field.Store.YES, Field.Index.UN_TOKENIZED,
    Field.TermVector.NO); //定义 id 重用列
Field nameField = new Field("cname", null ,
    Field.Store.YES, Field.Index.TOKENIZED,
    Field.TermVector.WITH_POSITIONS_OFFSETS);
//定义 name 重用列
while(rs.next()) {
    Document doc = new Document();
    idField.setValue(String.valueOf(rs.getInt("companyId")));
    doc.add(idField);
    nameField.setValue(rs.getString("companyName"));
    doc.add(nameField);
    index.addDocument(doc);
}

```

注意，不能在一个文档中重用单个 Field 实例，不应该改变一个列的值，直到包含这个 Field 的 Document 已经加入到索引库。

【作者提示】当在 Windows 系统下使用的时候。记住最好关闭杀毒软件的自动删除已感染病毒文件的选项。否则当索引带病毒特征的文档时，杀毒软件可能破坏 Lucene 的索引文件。图 6-15 是索引库文件被杀毒软件删除的例子。



图 6-15 杀毒软件删除索引库文件

每一个添加的文档都被传递给 DocConsumer 类，它处理该文档并且与索引链表 (indexing chain) 中其他的 consumers 相互发生作用。确定的 consumers (如 StoredFieldWriter 和 TermVectorsTermsWriter) 提取一个文档中的词，并且马上把字节写入文件。

IndexWriter.setRAMBufferSizeMB 方法可以设置更新文档使用的内存达到指定大小之后才写入硬盘，这样可以提高写索引的速度，尤其是在批量建索引的时候。



### 6.3.3 删除索引库中的索引文档

在 Lucene 2.1.0 之前，删除文档是在 `IndexReader` 中实现的。从 Lucene 2.1.0 开始，可以从 `IndexWriter` 中删除文档了。

```
indexWriter.delete(new Term("id", "1"));
```

`IndexReader` 和 `IndexWriter` 都能够进行文档删除，其中的区别是：当 `IndexWriter` 打开索引的时候，`IndexReader` 的删除操作会抛出 `LockObtainFailedException` 异常。

### 6.3.4 更新索引库中的索引文档

Lucene 早期的版本，先通过 `IndexReader` 删除文档，然后再通过 `IndexWriter` 增加文档。在 Lucene 2.1.0 以后 `IndexWriter` 直接提供了更新文档的接口。

```
indexWriter.updateDocument(new Term("url", "http://www.lietu.com"),  
document);
```

Lucene 的 `updateDocument` 方法仍然是先删除旧文档然后再向索引增加传入的新文档。如果只希望更新个别列而保持其他的列不动就会有问题。为了解决这个问题，可以搜索索引中的当前文档，改变要改变的列，然后把修改后的文档作为参数传给 `updateDocument`。

```
public void searchAndUpdateDocument(IndexWriter writer, IndexSearcher  
searcher, Document updateDoc, Term term) throws IOException {  
    TermQuery query = new TermQuery(term);  
  
    TopDocs hits = searcher.search(query, 10);  
  
    if (hits.scoreDocs.length == 0) {  
        throw new IllegalArgumentException("索引中没有匹配的结果");  
    } else if (hits.scoreDocs.length > 1) {  
        throw new IllegalArgumentException("Given Term matches  
more than 1 document in the index.");  
    } else {  
        int docId = hits.scoreDocs[0].doc;  
  
        //找出旧的文档  
        Document doc = searcher.doc(docId);  
  
        List<Field> replacementFields = updateDoc.getFields();  
        for (Field field : replacementFields) {  
            String name = field.name();
```

```

        String currentValue = doc.get(name);
        if (currentValue != null) {
            //替换列值
            Field = new Field(name, term.text());
            //删除旧列所有的出现
            doc.removeFields(name);

            //插入替换列
            doc.add(field);
        } else {
            //新加到列
            doc.add(field);
        }
    }

    //把旧文档更改后重新写入索引
    writer.updateDocument(term, doc);
}
}

```

### 6.3.5 索引的合并

索引很多文档的过程通常比较慢。为了加快索引速度，可以多台机器同时索引不同内容，然后合并。要合并的几个不同的索引结构要一致。下面的程序可以把多个目录下的索引合并到一个目录下：

```

IndexWriter writer = new IndexWriter(args[0], null, true);
writer.setMergeFactor(50); //参数越大，用到的内存越多。影响内存的使用。
//影响索引文件的数量
writer.setUseCompoundFile(false);
Directory[] dirs = new Directory[args.length - 1];
System.out.println("begin :"+args[0]);
for (int i=1 ;i<args.length;i++) {
    dirs[i-1]= FSDirectory.getDirectory( args[i], false);
    if (dirs[i-1]==null)
        System.out.println("Directory is null:"+i);
}
writer.addIndexes(dirs);
writer.close();

```

### 6.3.6 索引文件格式

Lucene 把和一个索引相关的文件全部放在一个目录下，其中既存储描述索引结构的元数据，又包含索引数据。每类信息放在不同后缀的二进制文件中。元数据包含以 **fnm** 为后





缀的**列信息文件**。列信息文件格式是：

字段数量, <字段名称, 字段的二进制位描述>

其中：字段数量表示索引中字段的数量；字段名称是用字符串表示的字段名称；字段的二进制位描述是个 byte 值和 int 值。byte 值用一个最低位表示是否索引这个字段。例如：

```
doc.add(new Field("text", "content", Field.Store.NO, Field.Index.TOKENIZED));
```

在列信息文件中存储成：

1, <content, 0x01>

**词典文件**以 tis 为文件后缀。这个文件中的词是按顺序存放的。首先按词对应的字段名称排序，然后按词的正文排序。词典文件格式是：

词的数量, <词, 词的文档频率>

其中，词用前缀压缩的方式存储，格式是“前缀的长度，后缀，字段编号”。词的文档频率指词在多少个文档中出现过。在排好序的词表中，前后两个词往往包括共同的前缀。前缀的长度变量表示与前一项相同的前缀的字数。例如，如果前一个词是“bone”，后一个词是“boy”，则前缀的长度值为 2，后缀值为“y”。

例如有两个文档，内容是：

文档 1: Penn State Football ...football

文档 2: Football players ... State

在索引中的存储形式是：

4, <<0, football, 1>, 2> <<0, penn, 1>, 1> <<1, layers, 1>, 1> <<0, state, 1>, 2>

词典文件太大，为了能把词信息完整地读入内存，设计出了词信息索引文件（.tii）。词信息索引文件不存储词本身，但是保存随机读取的文件的位置信息。

词在文档中出现的**频率文件**以 frq 为文件后缀。词频率首先按照 tis 中的词序来排列，在每个词内部，存储这个词在文档中出现的频率按照 docID 排序。频率文件中并没有存储

docID，而是存储与前后两个 docID 的增量相关的一个值 DocDelta。频率文件格式是：

<DocDelta[, Freq?]>

DocDelta 同时决定了文档号和频数。详细地说，DocDelta/2 表示当前 docID 相对于前一个 docID 的偏移量（或者是 0，表示这是 TermFreqs 里面的第一项）。当 DocDelta 是奇数时，表示在该文档中频数为 1；当 DocDelta 是偶数时，下一个整数表示在该文档中出现的频数。

例如，假设某一项在文档 7 中出现了一次，在文档 11 中出现了 3 次，在 TermFreqs 中就存在如下的整数序列：15, 8, 3。在这里：

15=2×7+1→在文档 7 中出现频率是 1

8=2×(11-7)→在文档 11 中出现频率>1

3→在文档 11 中出现频率=3

表 6-4 中的 Posting List 对应的频率文件存储内容是：

<<2, 2, 3> <3> <5> <3, 3>>

表 6-4 Posting List

Posting id	词	docID	offset
1	football	1	3
		1	67
		2	1
2	penn	1	1
3	players	2	2
4	state	1	2
		2	13

存储词在文档中出现过的位置的**位置文件**以 prx 为后缀。TermPositions 按照词来排序（依据.tis 文件中词的位置）。Positions 数值按照 docID 升序排列。实际存储的是 PositionDelta 值，PositionDelta 是当前位置相对于前一个出现位置（或者为 0，表示这是第一次在这个文档中出现）的增量值。例如，假设某词在某文档第 4 项出现，在接下来的一个文档中第 5 项和第 9 项出现，将表示为如下的整数序列：4, 5, 4。表 6-4 对应的位置文件存储内容是：

<<3, 64> <1>> <<1> <0>> <<0> <2>> <<2> <13>>

Lucene 的查询过程访问的文件如图 6-16 所示。

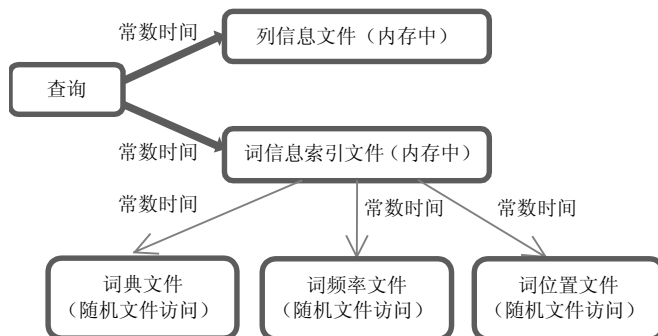


图 6-16 查询访问文件

复合文件格式的索引优化后只有 3 个文件，其中 `segments_N` 和 `segments.gen` 是固定不变的，因为这两个文件是在索引级别存在的文件，还有一个是复合索引文件格式（.cfs）。可以通过 `setUseCompoundFile` 方法设定是否使用复合文件格式。



## 6.4 查找索引库

可以按关键词查询指定的列，根据相关度返回结果，也可以自定义搜索结果排序方式。

### 6.4.1 查询过程

Lucene 中使用 `IndexSearcher` 类来对索引进行检索。`IndexSearcher` 类继承自抽象类 `Searcher`。查询依赖一个或者多个索引库，`IndexSearcher` 依赖一个或者多个 `IndexReader`。查询执行过程如图 6-17 所示。

在初始化一个 `IndexSearcher` 类时，最重要的就是要告诉它使用哪个索引完成查找任务。可以用 `IndexReader` 来初始化一个 `IndexSearcher` 对象。

```

Directory directory = FSDirectory.open(new File("E:/luceneTest/
fileindex"));
IndexReader indexReader = DirectoryReader.open(directory);
IndexSearcher indexSearcher = new IndexSearcher(indexReader);
  
```

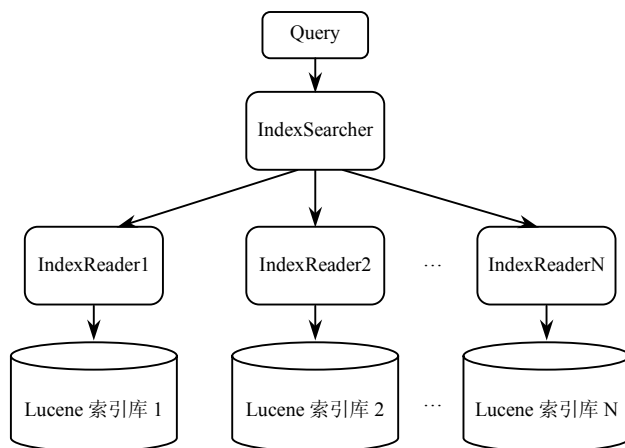


图 6-17 查询执行过程

`DirectoryReader` 是 `IndexReader` 的子类。使用 `DirectoryReader.open(directory)` 方法得到 `IndexReader` 的实例。

`Directory` 类型的对象包含了索引存放的路径信息，从而定位了索引。这里使用了 `Directory` 类型的对象来创建 `IndexReader`，然后再用 `IndexReader` 构建 `IndexSearcher`。要查询的索引往往只有一个，所以也可以直接使用 `Directory` 类型的对象来构建 `IndexSearcher`。

```
String indexDir = "D:/indexdir"; //索引库路径
//打开一个文件路径，只读的方式使用 directory，所以 read-only=true
Directory directory = FSDirectory.open(new File(indexDir));
//隐式创建了 IndexReader
IndexSearcher searcher = new IndexSearcher(directory, true);
```

这只是一个隐式创建 `IndexReader` 的简单写法。使用 `Directory` 构造的 `IndexSearcher` 实例仍然各自持有一个 `IndexReader` 实例，若系统中存在同一个索引的多个 `IndexSearcher` 实例时，将占用过多的内存空间。这时，应该是一份索引用一个 `IndexReader` 实例打开，从 `IndexReader` 构造 `IndexSearcher` 实例。

Lucene 各个不同版本的索引格式不完全一致，可以先用 `IndexReader.getVersion()` 取得版本号：

```
System.out.println("版本号 :"+reader.getVersion());
```

把 111.doc 分成 111，1111.doc 分成 1111，11111.doc 分成 11111。这样输入 111，就只能搜索到 111.doc。一段有意义的文字需要通过 `Analyzer` 分割成一个个词语后才能按关键词



搜索。Analyzer 就是分析器，StandardAnalyzer 是 Lucene 中最常用的分析器。

```
//指定 Lucene 版本号  
Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_43);
```

为了达到更好的搜索效果，不同的语言可以使用不同的分析器，例如 CnAnalyzer 就是一个主要处理中文的分析器。

Analyzer 返回的结果就是一串 Token。Token 包含一个代表词本身含义的字符串和该词在文章中相应的起止偏移位置，Token 还包含一个用来存储词类型的字符串。

Term 是搜索语法的最小单位，复杂的搜索语法会分解成一个个的 Term 查询。它表示文档的一个词语，Term 由两部分组成：它表示的词语和这个词语所出现的 Field。例如：

```
new Term("url", "http://www.lietu.com");
```

最简单的 Query 对象是 TermQuery。根据 TermQuery 查询索引库。

```
Query query = new TermQuery(new Term("url", "http://www.lietu.com"));  
  
//搜索索引库并且返回最相关的 10 个文档  
TopDocs tds = searcher.search(query, 10);
```

要是用户输入个逗号，也会搜出文档来？逗号是停用词。一般停用词进索引，但是搜索的时候滤掉。指用户输入逗号，系统识别为停用词就给提示说这个玩意不能搜。

查询串中可能包括一些高级查询语法，例如要找包含 java 的 pdf 文件，可以使用查询串“java filetype:pdf”。所以用查询分析器 QueryParser 来解析查询串，也就是根据查询串生成 Query 对象。例如：

```
//指定 Lucene 版本号  
Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_43);  
QueryParser qp = new QueryParser(Version.LUCENE_43, fields, analyzer);  
query = qp.parse(queryString);
```

使用 IndexSearcher 的 search 来执行搜索，返回一个 TopDocs 对象。TopDocs 对象中的 totalHits 属性记录了搜索返回结果总条数。基本的关键词查询代码如下：

```
String indexDir = "D:/indexdir"; //索引库路径  
//打开一个文件路径
```

```

Directory directory = FSDirectory.open(new File(indexDir));
// read-only=true
IndexSearcher searcher = new IndexSearcher(directory, true); //搜索

String defaultField = "title"; //默认查询列
String queryString ="NBA"; //查询词
//指定 Lucene 版本号
Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_43);
QueryParser parser = new QueryParser(Version.LUCENE_43,
                                     defaultField,
                                     analyzer);

Query query = parser.parse(queryString);
TopDocs docs = searcher.search(query, 10); //查询最多只返回前 10 条结果

```

这里的 `FSDirectory` 表示硬盘中的索引。`RAMDirectory` 表示内存中的索引，可以用来测试索引和查找过程。

遍历查询结果：

```

ScoreDoc[] hits = docs.scoreDocs; //从 TopDocs 取得查询结果

//遍历结果
for (ScoreDoc hit : hits){
    Document hitDoc = searcher.doc(hit.doc);
    //输出标题和文档相关度分值
    System.out.println(hitDoc.get("title") +hit.score);
}

```

`ScoreDoc` 中的 `score` 属性就是相关度。相关度得分是 1 到 0 之间的值。1 表示相关度最高，而 0 则表示不相关。文档的相关度跟很多因素有关。比如字段的长短，里面词条的权重等。决定 **SCORE** 的简单概括：1、项频率，即查询项在某个文档中出现的次数；2、文档频率，即查询项在很多文档数中出现的次数。

完整的查询代码：

```

Directory directory = FSDirectory.open(new File("d:/testindex"));
//DirectoryReader 读入一个目录下的索引文件
IndexReader ir = DirectoryReader.open(directory);
//打开索引库
IndexSearcher searcher = new IndexSearcher(ir);

```



```
//根据查询词搜索索引库
TopDocs docs = searcher.search(new TermQuery(new Term("title","text")),
10);
//遍历查询结果
ScoreDoc[] hits = docs.scoreDocs;
for (ScoreDoc hit : hits){
    System.out.print("doc:"+hit.doc+" score:"+hit.score+"\n");
}
//关闭索引库
ir.close();
```

## 6.4.2 常用查询

Query 是一个用于查询的抽象基类。有各种具体查询实现类，比如实现基本词查询的 TermQuery、实现布尔逻辑查询的 BooleanQuery、实现短语查询的 PhraseQuery、实现前缀匹配查询的 PrefixQuery、实现区间查询的 RangeQuery、实现多词查询的 MultiTermQuery、实现过滤条件查询的 FilteredQuery、约束多个查询词密集度的 SpanQuery 等。

可以使用 QueryParser 查询分词列。一般不需要查询使用没有分词的列，如果要查询可以使用 TermQuery 查询，例如“url”列。

最基本的词条查询使用 TermQuery，用来查询不切分的字段。例如查询一个杂志：

```
Term term = new Term("journal_id","1672-6251");
TermQuery query = new TermQuery(term);
```

为了方便测试查询，使用 createDocument 方法索引测试内容。

```
private static Document createDocument(String id, String content) { //
创建文档
    Document doc = new Document();
    doc.add(new Field("id", id, StringField.TYPE_STORED)); //索引不分词的字符串列
    //索引分词的文本列
    doc.add(new Field("contents", content, TextField.TYPE_STORED));
    return doc;
}
```

组合条件查询使用布尔逻辑查询 BooleanQuery。举一个布尔逻辑查询的例子：同时查询标题列和内容列。使用 BooleanClause.Occur.SHOULD 的查询效果如图 6-18 所示。

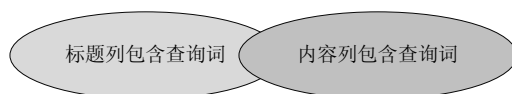


图 6-18 BooleanClause.Occur.SHOULD 效果

```

QueryParser parser = new QueryParser(Version.LUCENE_CURRENT, "body",
    analyzer);
Query bodyQuery = parser.parse("NBA");//查询内容列
parser = new QueryParser(Version.LUCENE_CURRENT, "title", analyzer);
Query titleQuery = parser.parse("NBA");//查询标题列

BooleanQuery bodyOrTitleQuery = new BooleanQuery();
//用 OR 条件合并两个查询
bodyOrTitleQuery.add(bodyQuery, BooleanClause.Occur.SHOULD);
bodyOrTitleQuery.add(titleQuery, BooleanClause.Occur.SHOULD);

//返回前 10 条结果
ScoreDoc[] hits = isearcher.search(bodyOrTitleQuery, 10).scoreDocs;

```

### 6.4.3 基本词查询

TermQuery 是最简单、也是最基本的 Query。TermQuery 可以理解成为“词条搜索”，在搜索引擎中最基本的搜索就是在索引中搜索某一词条，而 TermQuery 就是用来完成这项工作的。

查询某个字段中所包含的某个关键字。例如，查询类别列是服装的商品。

```

Query query = new TermQuery(new Term("cat", "服装"));

//搜索索引库并且返回最相关的 10 个商品
TopDocs tds = searcher.search(query, 10);

```

### 6.4.4 模糊匹配

关于字符串的前缀匹配问题：只查询产品名字为字母 A 打头的搜索，可以使用前缀匹配(PrefixQuery)实现。查询语法是 A\*。注意，查询的不一定是整个字段以 A 开头的纪录，而只是其中的单词以 A 开头。

```

Query query = new WildcardQuery(new Term(FIELD, "cut*")); //通配符

```

等价于：

```

Query query = new PrefixQuery(new Term(FIELD, "cut")); //自动在结尾添加 *

```





测试 WildcardQuery:

```
//索引一些文档
writer.addDocument(createDocument("1", "foo bar baz"));
writer.addDocument(createDocument("2", "red green blue"));
writer.addDocument(createDocument("3",
    "The Lucene was made by Doug Cutting"));
writer.close();

IndexReader reader = DirectoryReader.open(directory);

IndexSearcher searcher = new IndexSearcher(reader);

Query query = new WildcardQuery(new Term(FIELD, "cut*"));

TopDocs topDocs = searcher.search(query, 10);
```

这里的 `cut` 只能是小写，否则就匹配不上。如果用户输入的是大写的 `Cut`，可以用 `QueryParser` 转换成小写。

使用反转 `Token` 的方法允许通配符出现在开头。`ReversedWildcardsTokenFilter` 实现 `Token` 反转。`ReversedWildcardsTokenFilter` 返回原来的 `Token` 和反转的 `Token`，其中反转的 `Token` 的 `positionIncrement` 值是 0。

用一个标识字符来避免正常 `Token` 和反转 `Token` 之间的冲突。例如，"`DNA`" 反转后就变成"`and`"了，和正常的词"`and`"有冲突，但是用标识字符后，"`DNA`" 变成了"`\u0001and`"。

可以把这个 `TokenFilter` 加到分析器链，这样在做索引的时候就可以使用 `ReversedWildcardsTokenFilter`。

有些英文单词需要查询扩展。例如搜索“`dog`”的同时查找“`dogs`”。`FuzzyQuery` 有限状态查询(Finite-State Query)用编辑距离衡量相似度，例如 `dog` 和 `dogs` 的编辑距离是 1。`FuzzyQuery` 内部使用编辑距离有限状态机实现，所以性能很好。

```
int maxEdits = 2; //编辑距离最多不能超过 2
new FuzzyQuery(new Term("title", "dog"), maxEdits);
```

可以设置相同前缀的长度。例如，相同前缀的长度是 1:

```
FuzzyQuery query = new FuzzyQuery(new Term("field", "WEBER"), 2, 1);
```

query 要求匹配出来的词必须以 W 开头，而且匹配出来的词与查询词之间的编辑距离不超过 2。这个 FuzzyQuery 等价于如下的有限状态自动机：

```
LevenshteinAutomata builder = new LevenshteinAutomata("EBER", true);
Automaton a = builder.toAutomaton(2); //最大编辑距离是 2
Automaton b = BasicAutomata.makeChar('W'); //创建一个字符 w 组成的自动机
Automaton c = BasicOperations.concatenate(b, a); //连接两个自动机 b 和 a
```

这里的 BasicOperations.concatenate 方法把 b 的结束状态和 a 的开始状态之间用空转换连接起来，也就是按顺序走过 a 中的状态，然后走 b 中的状态。例如自动机 b 可以接收字符 W，然后自动机 a 继续接收 EB，就可以结束了。这说明自动机 c 可以接收 WEB。

测试自动机 c 可以接收哪些字符：

```
System.out.println(BasicOperations.run(c, "WBR")); //输出 true
System.out.println(BasicOperations.run(c, "WEB")); //输出 true
System.out.println(BasicOperations.run(c, "WEBE")); //输出 true
System.out.println(BasicOperations.run(c, "WEBER")); //输出 true
```

在中文人名中使用自动机：

```
LevenshteinAutomata builder = new LevenshteinAutomata("杰伦", true);
Automaton a = builder.toAutomaton(1); //最大编辑距离是 1
Automaton b = BasicAutomata.makeChar('周');
Automaton c = BasicOperations.concatenate(b, a); //连接两个自动机 b 和 c
System.out.println(BasicOperations.run(c, "周杰轮"));
```

在中文人名中使用模糊匹配：

```
FuzzyQuery query = new FuzzyQuery(new Term("field", "周杰伦"), 1, 1);
```

“dogs~” 这样的模糊查询语法使用 FuzzyQuery。FuzzyQuery 还可以用于拼写检查。

Lucene 的 NumericRangeQuery 采用了 Trie 树结构的索引，可以模仿 NumericRangeQuery 写个字符串的前缀匹配实现。

### 6.4.5 布尔查询

可以使用 BooleanQuery 组合多个查询条件。合取查询使用 BooleanClause.Occur.MUST 连接多个查询词。



```

Term t1 = new Term(FIELD, "lucene");
TermQuery q1 = new TermQuery(t1);

Term t2 = new Term(FIELD, "doug");
TermQuery q2 = new TermQuery(t2);

//合取查询
BooleanQuery query = new BooleanQuery();
query.add(q1, BooleanClause.Occur.MUST); //必须包含这个条件
query.add(q2, BooleanClause.Occur.MUST);

```

选择出包含任何一个查询词的文档叫做析取(Disjunction)。用真值表描述析取。

p	q	(p ∨ q)
T	T	T
T	F	T
F	T	T
F	F	F

析取查询使用 `BooleanClause.Occur.SHOULD` 连接多个查询词。

```

//析取查询
BooleanQuery q = new BooleanQuery();
q.Add(q1, BooleanClause.Occur.SHOULD); //可以只包含这个条件
q.Add(q2, BooleanClause.Occur.SHOULD);

```

去餐馆点菜,有人不喜欢吃辣的东西,所以他找出所有不辣的菜。这叫做否定(negation)。用真值表描述否定。

p	~p
T	F
F	T

`BooleanClause.Occur.MUST_NOT` 表示不能包括符合这个条件的文档。例如找出不包含“辣”的所有文档。

```

Term t1 = new Term("body", "辣");
TermQuery q1 = new TermQuery(t1);

BooleanQuery mbq = new BooleanQuery();
MatchAllDocsQuery alldocs = new MatchAllDocsQuery(); //匹配所有文档
mbq.Add(q1, BooleanClause.Occur.MUST_NOT); //不包括满足 q1 条件的文档
mbq.Add(alldocs, BooleanClause.Occur.MUST);

```

打麻将三缺一不行，但是如果同时查询多个词，最好能匹配上只差一个查询词的文档。

对于长句搜索，则提取其中的主要查询词，只要大部分词在文档中出现就可以了。例如，搜索联系方式列：

```
- "Stanford University School of Medicine, Palo Alto, CA USA",
- "Institute of Neurobiology, School of Medicine, Stanford University,
Palo Alto, CA",
- "School of Medicine, Harvard University, Boston MA",
- "Brigham & Women's, Harvard University School of Medicine, Boston, MA"
- "Harvard University, Cambridge MA"
```

搜索联系方式使用如下的长查询词：

```
"School of Medicine, Stanford University, Palo Alto, CA"
```

需要找到所有和斯坦福相关的文档。

**PhraseQuery** 要求短语中所有的词都存在才能匹配上。需要一个更加宽松的 **PhraseQuery** 版本，它对词出现的顺序敏感，但是允许缺少个别词，缺少词的文档分值低，但是仍然能匹配上。**BooleanQuery.setMinimumNumberShouldMatch(int)** 方法可以定义需要匹配上的条件的最小数量。

例如，**BooleanQuery** 中有 2 项。调用 **BooleanQuery.setMinimumNumberShouldMatch(1)** 方法可以匹配任意其中的 1 项或者更多。

```
Term t1 = new Term(FIELD, "鸡");
TermQuery q1 = new TermQuery(t1);

Term t2 = new Term(FIELD, "鸭");
TermQuery q2 = new TermQuery(t2);

BooleanQuery mbq = new BooleanQuery();
mbq.add(q1, BooleanClause.Occur.SHOULD);
mbq.add(q2, BooleanClause.Occur.SHOULD);
mbq.setMinimumNumberShouldMatch(1);
```

#### 6.4.6 短语查询

如果按字索引中文文档，则查询的时候往往要求文档中的这些字是紧密相联的。查询紧密相联的几个关键词。**PhraseQuery** 叫做短语匹配查询，用于要求精确匹配的查询。



PhraseQuery 使用的词保存在索引中的位置信息，因此需要索引中的相关列已经保存了位置信息。例如按字索引，按字查询“开封”：

```
PhraseQuery query = new PhraseQuery();
query.add(new Term("subject", "开"));
query.add(new Term("subject", "封"));
```

有时候要匹配上的词之间有间隔，匹配上的词之间的距离称为 slop。默认情况下，slop 值是零，可以通过调用 setSlop 方法设置这个值。例如用户搜索“西红柿牛腩”匹配上“西红柿炖牛腩”。

```
PhraseQuery query = new PhraseQuery();
query.setSlop(1);
query.add(new Term("subject", "西红柿"));
query.add(new Term("subject", "牛腩"));
```

使用 PhraseQuery 完整的例子：

```
public static void main(String[] args) throws Exception {
    //在内存中建立索引
    Directory directory = new RAMDirectory();
    Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_43);
    IndexWriterConfig iwc = new IndexWriterConfig(Version.LUCENE_43,
        analyzer);
    IndexWriter writer = new IndexWriter(directory, iwc);

    //索引一些文档
    writer.addDocument(createDocument("1", "foo bar baz"));
    writer.addDocument(createDocument("2", "red green blue"));
    writer.addDocument(createDocument("3", "test foo bar test"));
    writer.close();

    //查找包含"foo bar"这个短语的文档
    String sentence = "foo bar";
    IndexReader reader = IndexReader.open(directory);
    //根据 IndexReader 创建 IndexSearcher
    IndexSearcher searcher = new IndexSearcher(reader);
    PhraseQuery query = new PhraseQuery();
    String[] words = sentence.split(" ");
    for (String word : words) {
        query.add(new Term("contents", word));
    }
}
```

```

//显示搜索结果
TopDocs topDocs = searcher.search(query, 10);
for (ScoreDoc scoreDoc : topDocs.scoreDocs) {
    Document doc = searcher.doc(scoreDoc.doc);
    System.out.println(doc);
}

private static Document createDocument(String id, String content) {
    Document doc = new Document();
    doc.add(new Field("id", id, Store.YES, Index.NOT_ANALYZED));
    doc.add(new Field("contents", content, Store.YES, Index.ANALYZED,
        Field.TermVector.WITH_POSITIONS_OFFSETS));
    return doc;
}

```

### 6.4.7 跨度查询

查询词是“吃饭”，文档中出现了“吃完饭”。把查询词和文档都按最小粒度分词，分成“吃”和“饭”两个词。文档中的这两个词虽然不是连续出现，但只间隔了一个词。

近似查询假设同时查询的几个词之间的匹配点很近，则这样的文档可能是要找的。匹配词在文档中的位置信息用 **Spans** 描述。**Spans** 封装了一次匹配的文档和位置信息。一个 **span** 是一个<文档编号，开始位置，结束位置>的三元组。

它的接口定义如下：

```

boolean next()           //移动到下一个匹配的文档
boolean skipTo(int target) //跳到指定的文档
int doc()                //当前的文档编号
int start()              //匹配区域的开始位置
int end()                 //匹配区域的结束位置

```

**TermSpans** 是 **Spans** 的具体子类。

**SpanQuery** 叫做跨度查询，用于查询多个词时考虑几个词在文档中的匹配位置。**SpanQuery** 和 **PhraseQuery**s 或者 **MultiPhraseQuery**s 很相似，因为都通过位置限制匹配。但是 **SpanQuery** 更灵活。

**SpanNearQuery** 用来查询在比较近的区域内出现的多个查询词。例如下面这个查询既可以匹配上“吃饭”，又可以匹配上“吃完饭”。

```
new SpanNearQuery(new SpanQuery[] {
    new SpanTermQuery(new Term(FIELD, "吃")), //第一个词
    new SpanTermQuery(new Term(FIELD, "饭"))}, //第二个词
    1, //在 1 个位置以内
    true); //有序
```

`SpanTermQuery` 是一个最基础的跨度搜索实现类，因为可以通过它得到一个词的位置信息。在 `SpanNearQuery` 的构造方法中指定一些查询词要在多近的区域出现。

都是那么几个词在一起，但是如果顺序不一样，意思可能就不一样，例如“花的中间”与“中间的花”的意义就不一样。所以，`SpanNearQuery` 有一个参数表示是否要求匹配按数组中的顺序。

例如，需要查找 `lucene` 和 `doug` 在 5 个位置以内。`doug` 在 `lucene` 之后，也就是说，词出现的顺序是需要考虑的。可以使用如图 6-19 所示的 `SpanQuery`。

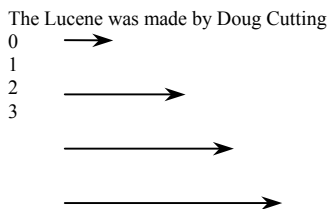


图 6-19 `SpanNearQuery`

```
SpanQuery[] baseQueries = new SpanQuery[] {
    new SpanTermQuery(new Term(FIELD, "lucene")),
    new SpanTermQuery(new Term(FIELD, "doug"))};
//这里的 true 表示词必须按数组中的顺序出现在文档中
new SpanNearQuery(baseQueries, 5, true);
```

在这个例子中，`Lucene` 和 `Doug` 的间隔在 3 以内，也就是间隔了 3 个词。

`SpanNearQuery` 会查找互相在给定距离内的一些数量的 `SpanQuery`。可以要求 `span` 是按顺序的或者不需要考虑顺序。`SpanNearQuery` 根据 `SpanQuery` 对象组成的数组来构建。因为 `SpanTermQuery` 是 `SpanQuery` 的子类，所以可以根据 `SpanTermQuery` 构造出 `SpanNearQuery`。

`SpanNearQuery` 构造方法接收一个 `SpanQuery` 的数组。`span` 之间的距离，一个布尔值表明是否要求按 `SpanQuery` 数组中的顺序出现。如图 6-20 所示是另外一个嵌套的例子。这

次，查找 `doug` 在 `lucene` 之后的 5 个间隔内，然后 `hadoop` 在 `lucene -> doug span` 之后的 4 个间隔内。

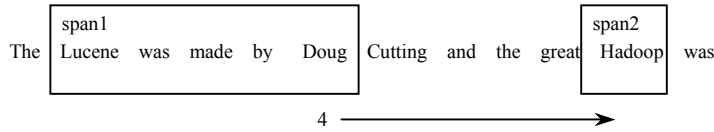


图 6-20 嵌套的 SpanNearQuery

```
SpanNearQuery spanNear = new SpanNearQuery(new SpanQuery[] {
    new SpanTermQuery(new Term(FIELD, "lucene")),
    new SpanTermQuery(new Term(FIELD, "doug"))},
    5,
    true);

new SpanNearQuery(new SpanQuery[] {
    spanNear,
    new SpanTermQuery(new Term(FIELD, "hadoop"))},
    4,
    true);
```

有些列，例如标题，头几个词可能更重要，可以使用 `SpanFirstQuery` 限定只查询前几个词。可以把 `SpanFirstQuery` 看成是 `SpanNearQuery` 的特例，第一个词是一个虚拟的开始词，从第二个词开始才是真正的匹配词。

例如有两个文档 `"I love Lucene"` 和 `"Lucene is nice"`，想要能够查询 `Lucene` 出现在最开始的文档，也就是匹配正则表达式 `^Lucene .*` 的文档。

```
SpanTermQuery lucene = new SpanTermQuery(new Term("title", "Lucene"));
//限定在头一个位置出现
SpanFirstQuery first = new SpanFirstQuery(lucene, 1);
```

`SpanRegexQuery` 使用标准的正则表达式语法。例如想要查询年，2000, 2001, 2002, ... 2009。

```
SpanRegexQuery srq = new SpanRegexQuery(new Term("year", "200.?"));
```

`SpanNotQuery` 包含必须满足的 `SpanQuery` 和必须排除的 `SpanQuery`。例如，想要找 `"Microsoft Windows"`，但是不匹配 `"Microsoft Windows"` 之后的文档是大写的，伪代码如下：

```
SpanNot:
    include:
```





```
SpanNear(in-order=true, slop=0):
    SpanTerm: "Microsoft"
    SpanTerm: "Windows"
exclude:
    SpanNear(in-order=true, slop=0):
        SpanTerm: "Microsoft"
        SpanTerm: "Windows"
        SpanRegex: "^\\p{Lu}.*"
```

实际的代码：

```
SpanNearQuery includeQuery = new SpanNearQuery(new SpanQuery[] {
    new SpanTermQuery(new Term(FIELD, "Microsoft")),
    new SpanTermQuery(new Term(FIELD, "Windows"))},
    0,
    true);

SpanRegexQuery upperCaseQuery = new SpanRegexQuery( new Term(FIELD,
"^\\p{Lu}.*"));

SpanNearQuery excludeQuery = new SpanNearQuery(new SpanQuery[] {
    new SpanTermQuery(new Term(FIELD, "Microsoft")),
    new SpanTermQuery(new Term(FIELD, "Windows")),
    upperCaseQuery },
    0,
    true);
```

或者想找包括“喜欢”的文档，而不包括“不”字在前面的。可以这样：

```
SpanNotQuery(like, SpanNearQuery(not, like))
```

再扩展出同类查询词，也就是：

```
SpanNotQuery(
    SpanOrQuery(喜欢, 爱...)
    SpanNearQuery(
        SpanOrQuery(不)
        SpanOrQuery(喜欢, 爱...)
    )
)
```

SpanOrQuery 可以匹配符合任何一个条件的文档：

```
SpanTermQuery term1 = new SpanTermQuery(new Term("field", "thirty"));
SpanTermQuery term2 = new SpanTermQuery(new Term("field", "three"));
```

```
SpanNearQuery near1 = new SpanNearQuery(new SpanQuery[] {term1, term2},
                                          0, true);
SpanTermQuery term3 = new SpanTermQuery(new Term("field", "forty"));
SpanTermQuery term4 = new SpanTermQuery(new Term("field", "seven"));
SpanNearQuery near2 = new SpanNearQuery(new SpanQuery[] {term3, term4},
                                          0, true);

SpanOrQuery query = new SpanOrQuery(new SpanQuery[] {near1, near2});
```

例如说一个人坏透了，用“头上长疮，脚下流脓”来形容。这个条件可以这样描述：

```
SpanAndQuery(
    SpanFirstQuery(疮)
    SpanLastQuery(脓)
)
```

SpanAndQuery 和 SpanLastQuery 都还没有现成的实现。

#### 6.4.8 FieldScoreQuery

FieldScoreQuery 叫做函数查询（通过数字型的字段影响排序结果），时间加权排序时会用到。

除了文本列上的使用 tf-idf 相似性的标准的词查询之外，打分还参考数值列的相似性。相似性依赖于查询对象中的值和文档中的数值列的值之间的距离。（例如，高斯函数，使用参数：m=[user input], s=0.5）。

例如，猎头使用搜索引擎找人，表示人的文档有两列：

```
description (文本列)
age (数值列)
```

想要找这样的文档：

```
description:(x y z) age:30
```

但是 age 不是过滤条件，而是 score 的一部分。（对于 30 岁的人，乘积因子是 1.0，对 25 岁的人，乘积因子是 0.8，等等）

把 ValueSourceQuery 和 TermQuery 包装在 CustomScoreQuery 中实现。

```
public class AgeAndContentScoreQueryTest extends TestCase{
```



```
public class AgeAndContentScoreQuery extends CustomScoreQuery {
    protected float peakX;
    protected float sigma;

    //接收 4 个参数，其中 subQuery 表示文本列查询，valSrcQuery 表示值查询
    public AgeAndContentScoreQuery(Query subQuery, ValueSourceQuery
valSrcQuery, float peakX, float sigma) {
        super(subQuery, valSrcQuery);
        this.setStrict(true); //不归一化 ValueSourceQuery 中的分值
        this.peakX = peakX;    //年纪相关度中哪个年纪是最好的
        this.sigma = sigma;
    }

    @Override
    public float customScore(int doc, float subQueryScore, float
valSrcScore){
        // subQueryScore 是来源于内容查询的 td-idf 分值
        float contentScore = subQueryScore;

        // valSrcScore 是一个出生日期列的值，表示成浮点数
        // 把年纪值转换成高斯分布那样的年纪相关度分析值
        float x = (2011 - valSrcScore); // 年纪值
        float ageScore = (float) Math.exp(-Math.pow(x - peakX, 2) /
2*sigma*sigma);

        float finalScore = ageScore * contentScore;

        System.out.println("#contentScore: " + contentScore);
        System.out.println("#ageValue:      " + (int)valSrcScore);
        System.out.println("#ageScore:      " + ageScore);
        System.out.println("#finalScore:    " + finalScore);
        System.out.println("+++++++");

        return finalScore;
    }
}

protected Directory directory;
protected Analyzer analyzer = new WhitespaceAnalyzer();
protected String fieldNameContent = "content";
protected String fieldNameDOB = "dob";

protected void setUp() throws Exception {
    directory = new RAMDirectory();
}
```

```

analyzer = new WhitespaceAnalyzer();

// 索引文档
String[] contents = {"foo baz1", "foo baz2 baz3", "baz4"};
int[] dobs = {1991, 1981, 1987}; // 出生日期

IndexWriter writer = new IndexWriter(directory, analyzer,
IndexWriter.MaxFieldLength.UNLIMITED);
for (int i = 0; i < contents.length; i++) {
    Document doc = new Document();
    doc.add(new Field(fieldNameContent, contents[i],
Field.Store.YES, Field.Index.ANALYZED)); // 存储并索引
    doc.add(new NumericField(fieldNameDOB, Field.Store.YES,
true).setIntValue(dobs[i])); // 存储并索引
    writer.addDocument(doc);
}
writer.close();
}

public void testSearch() throws Exception {
    String inputTextQuery = "foo bar";
    float peak = 27.0f;
    float sigma = 0.1f;

    QueryParser parser = new QueryParser(Version.LUCENE_43,
fieldNameContent, analyzer);
    Query contentQuery = parser.parse(inputTextQuery);

    ValueSourceQuery dobQuery = new ValueSourceQuery( new
IntFieldSource(fieldNameDOB) );
    // 或者写成: FieldScoreQuery dobQuery = new FieldScoreQuery
(fieldNameDOB, Type.INT);

    CustomScoreQuery finalQuery = new AgeAndContentScoreQuery
(contentQuery, dobQuery, peak, sigma);

    IndexSearcher searcher = new IndexSearcher(directory);
    TopDocs docs = searcher.search(finalQuery, 10);

    System.out.println("\nDocuments found:\n");
    for (ScoreDoc match : docs.scoreDocs) {
        Document d = searcher.doc(match.doc);
        System.out.println("CONTENT: " + d.get(fieldNameContent) );
        System.out.println("D.O.B.: " + d.get(fieldNameDOB) );
        System.out.println("SCORE: " + match.score );
    }
}

```



```

        System.out.println("-----");
    }
}

```

日期加权：

```

if (x >= now) {
    score = now / x;
} else {
    score = (float) (now / (now+sigma) - sigma / x);
}

```



## 6.5 读写并发控制

在一个时刻只能够有一个线程修改索引库。Lucene 通过锁文件控制并发访问。在 Lucene 2.1 版本以后，控制写入的锁文件 `write.lock` 默认存储在 `index` 路径。

如果出现多余的锁文件，则有可能会抛出“Lock obtain timed out”异常。如果确定没有线程在修改索引，可以手工删除 `write.lock` 文件。

例如当全文索引放在只读光盘中时，需要设置这个索引只读。只读索引的初始化设置如下所示：

```

Directory indexDir =
    FSDirectory.getDirectory(indexPath, NoLockFactory.getNoLockFactory());

```



## 6.6 检索模型

检索模型也就是文档和查询词的相关度的评分方法。相关度评分在 Lucene 中由 `Similarity` 的子类实现。首先看一下如何定制 `Similarity`，然后再介绍常用的几种检索模型。

扩展 `DefaultSimilarity` 的一个例子如下所示：

```

private class SimilarityOne extends DefaultSimilarity {
    public float lengthNorm(String fieldName, int numTerms) {
        return 1;
    }
}

```

```
    }
}
```

然后在 IndexWriter 引用 SimilarityOne。

```
IndexWriter iw = new IndexWriter(dir,anlzh,true);
iw.setMaxBufferedDocs(5);
iw.setMergeFactor(3);
iw.setSimilarity(similarityOne);
iw.setUseCompoundFile(true);
...
iw.close();
```

也可以在 IndexSearcher 对象中指定自定义的 Similarity。

```
String indexDir = "/home/index";
IndexReader reader = IndexReader.open(indexDir);
IndexSearcher searcher=new IndexSearcher(reader);
searcher.setSimilarity(similarityOne);
```

### 6.6.1 向量空间模型

Lucene 使用布尔模型来确定哪些文档匹配上查询词,使用向量空间模型(VSM)来对这些文档评分。评分算法中的向量空间模型使用 Tf-idf 计算权重。对给定的词  $t$  和文档(或者查询)  $x$ ,  $Tf(t,x)$  的值和词  $t$  在  $x$  中出现的次数正相关,而  $idf(t)$  的值和索引文档集合中包含词  $t$  的次数负相关。

文档  $d$  对于查询词  $q$  的 VSM 分值是带权重的查询向量  $V(q)$  和文档向量  $V(d)$  的夹角余弦(Cos)相似度:

$$\text{cosine-similarity}(q,d) = \frac{V(q) \cdot V(d)}{\|V(q)\| * \|V(d)\|}$$

其中:

- 查询向量  $V(q) = \langle w(t_1, q), w(t_2, q), \dots, w(t_n, q) \rangle$ ;
- 文档向量  $V(d) = \langle w(t_1, d), w(t_2, d), \dots, w(t_n, d) \rangle$ ;
- $V(q) \cdot V(d)$  是两个带权重的向量的点积, 计算方法是  $V(q) \cdot V(d) = w(t_1, q) * w(t_1, d) + w(t_2, q) * w(t_2, d) + \dots + w(t_n, q) * w(t_n, d)$ ;
- $\|V(q)\|$  和  $\|V(d)\|$  表示欧几里得范数。例如  $\|V(d)\| = \sqrt{t_1^2 + t_2^2 + \dots + t_n^2}$ , 这里的  $t$  是文档  $d$  中出现的词的权重。

举个例子，共有 11 个词，共有 3 篇文档搜索出来。其中各自的权重（Term weight），如表 6-5 所示。

表 6-5 词项-文档矩阵表

	$t_1$	$t_2$	$t_3$	$t_4$	$t_5$	$t_6$	$t_7$	$t_8$	$t_9$	$t_{10}$	$t_{11}$
$D_1$	0	0	.477	0	.477	.176	0	0	0	.176	0
$D_2$	0	.176	0	.477	0	0	0	0	.954	0	.176
$D_3$	0	.176	0	0	0	.176	0	0	0	.176	.176
$Q$	0	0	0	0	0	.176	0	0	.477	0	.176

计算 3 篇文档同查询语句的相关性分值：

$$\text{SCORE}(D_1, Q) = \frac{0.176^2}{\sqrt{0.477^2 + 0.477^2 + 0.176^2 + 0.176^2} * \sqrt{0.176^2 + 0.477^2 + 0.176^2}} \approx 0.08$$

$$\text{SCORE}(D_2, Q) = \frac{0.954 * 0.477 + 0.176^2}{\sqrt{0.176^2 + 0.477^2 + 0.954^2 + 0.176^2} * \sqrt{0.176^2 + 0.477^2 + 0.176^2}} \approx 0.825$$

$$\text{SCORE}(D_3, Q) = \frac{0.176^2 + 0.176^2}{\sqrt{0.176^2 + 0.176^2 + 0.176^2 + 0.176^2} * \sqrt{0.176^2 + 0.477^2 + 0.176^2}} \approx 0.327$$

计算结果是  $D_2$  的相关性最高，其次是  $D_3$ ，最后是  $D_1$ 。

Lucene 通过以下方法改进了 VSM 评分公式来提高搜索结果相关性和实际可用性。

- 归一化向量  $V(d)$  成单位向量是有问题的，因为去除了所有的文档长度信息。对一些特定的文档可以归一化成为单位向量，比如，某些含有 10 次重复段落的文档，尤其是如果段落是由不同索引词组成的文档。但是对包含不重复段落的文档，应当归一化成为大于单位向量的向量。为了避免这个问题，对不同的文档使用不同的文档长度归一化因子  $\text{doc-len-norm}(d)$ ，这样把向量归一化成等于或者大于单位向量。为了节省计算时间，在索引文档语料的时候计算了  $\text{lengthNorm}(d)$ ，因此  $\text{lengthNorm}(d)$  并没有包含文档语料长度的统计信息。后面介绍的 BM25 算法在这方面有改进。
- 在索引阶段，用户可以通过给某个文档加权（boost）来调整文档的重要程度。所以，每个文档的分值要乘以它的加权值  $\text{doc-boost}(d)$ 。下面的代码设置整个文档的加权：

```
doc.setBoost(1.5F);
```

- Lucene 是按列搜索的，因此每个查询词应用于一个独立的列，文档长度归一化是按

索引列的长度来计算的，而且除了可以对文档整个加权，还可以对文档中不同的列设置不同的加权。下面的代码对标题列加权：

```
Field bodyField = new Field("body", body,
                             Field.Store.YES,
                             Field.Index.ANALYZED);
Field titleField = new Field("title", title,
                              Field.Store.YES,
                              Field.Index.ANALYZED);
titleField.setBoost(1.2F);
```

- 同样的列可以多次加入到文档，因此对一个列的加权是对文档中该列的多个值的倍数。
- 在搜索时，用户可以对每个查询、子查询和每个查询项说明加权值，因此用查询词的加权值  $\text{query-boost}(q)$  倍增查询词对文档分值的贡献度。
- 一个文档可能匹配上一个多词查询，但是可能不包含查询中所有的词。用户可以通过一个协调因子  $\text{coord-factor}(q,d)$  进一步奖励匹配更多查询词的文档，当匹配上更多的词时， $\text{coord-factor}(q,d)$  的值更大。如果要修改这个值，则不用重建索引就可以生效。

Lucene 的概念评分公式如下所示：

$$\text{score}(q, d) = \text{coord-factor}(q, d) \cdot \text{query-boost}(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)| |V(d)|} \cdot \text{doc-len} \\ - \text{norm}(d) \cdot \text{doc-boost}(d)$$

Similarity 类的实际计算公式：

$$\text{score}(q, d) = \text{coord}(q, d) * \text{queryNorm}(q) * \sum_{t \in q} (\text{tf}(t \text{ in } d) * \text{idf}(t)^2 * t.\text{getBoost()} * \text{norm}(t, d))$$

其中， $\text{tf}(t \text{ in } d)$  是  $t$  在文档  $d$  中的词频，默认实现是： $\sqrt{t}$  在文档  $d$  中的原始词频。

$\text{idf}(t)$  是词  $t$  在整个文档库中的倒文档频率，默认实现是： $\ln[N_{\text{Docs}} / (\text{docFreq} + 1)] + 1$ 。

通过 `org.apache.lucene.search.Explanation` 中提供的方法可以查看某个文档的得分的具体构成。

Lucene 的打分算法所涉及的因素如下：





- $tf$  = 词频 = 度量一个文档里词出现的频率
- $idf$  = 反向文档频率 = 度量一个词出现在索引中的频率
- $coord$  = 文档中发现的查询词的频率
- $lengthNorm$  = 根据索引列中的词总数来衡量一个词的重要度
- $queryNorm$  = 归一化的参数便于比较查询
- $boost(index)$  = 索引时的域的加权
- $boost(query)$  = 查询时的域的加权

前 4 项因素的实现、含义和原理说明如下。

### 1. $tf$

实现函数： $\sqrt{freq}$

含义：一个词在文档里出现的频率越高，则该文档的分值越高。

原理：多次包含同一个词的文档的相关度更高。

### 2. $idf$

实现函数： $\log(numDocs/(docFreq+1)) + 1$

含义：一个词越多出现在不同的文档里，则它的分值越低。

原理：常见词的重要性要低于不常出现的词。

### 3. $coord$

实现函数： $overlap / maxOverlap$

含义：对于查询中的词，文档中包含这些词越多分值就越高。

原理：分值高的文档要更多地覆盖查询中的词。

### 4. $lengthNorm$

实现： $1/\sqrt{numTerms}$

含义：如果匹配上包含较少词的索引列中的词，则这个文档有较高的权重。

**原理：**如果一个词在含有少量词的列中，则它比在包含较多词的列中的词更重要。

`queryNorm` 和文档的相关性无关，只是为了让不同的查询之间的分值有可比性。`queryNorm` 的实现函数是： $1/\sqrt{\text{sumOfSquaredWeights}}$ 。

因此，大致上说：

- 包含所有搜索词的文档是好的；
- 匹配很少出现的词比常用词更好；
- 长文档不如短文档好；
- 多次提及搜索词的文档更好。

有个问题是，Lucene 的默认的长度归一化公式对长的文档打分太低。想象在打羽毛球或网球，球拍面中有个最佳击球区，叫做甜区（sweet spots），只要在甜区范围内的回球都同样好。`SweetSpotSimilarity` 根据这个思想来改进长度归一化公式。`SweetSpotSimilarity` 实现了一个 `lengthNorm` 把一段区间内的文档长度看成同样好。可以定义一个全局的 `min/max`，在这段区间内的 `lengthNorm` 都是 1.0。低于最小值或高于最大值的 `lengthNorm` 以一个平方的函数下降。这样的结果是，在此区间内稍微长点的文档就不会有罚分了。

也可以对每列设置不同的 `min/max`，这样它们有不同的甜区。选择 `min/max` 可以根据文档的平均长度。你必须提前知道和推测出这个平均长度，并且通过方法 `SweetSpotSimilarity.setLengthNormFactors(yourAvg,yourAvg,steepness)` 把它明确地设定为优选区。比如要做个论文的搜索系统，经过统计发现大多数论文的长度在 8000 到 10000 词，因而 `min` 值设为 8000，`max` 值设为 10000。

做好这个 `SweetSpotSimilarity` 之后，用它来替换默认的 `Similarity`。在 `org.apache.lucene.search.Searcher` 中有个方法可以指定用哪个 `Similarity`：

```
public void setSimilarity(Similarity similarity) {
    this.similarity = similarity;
}
```

Nutch 中的 `NutchSimilarity` 类是一个计算网页相关性的例子。

### 6.6.2 BM25 概率模型

Okapi BM25（简称 BM25）是一种相关性排序函数，适用于搜索引擎根据与给定搜索查询的相关性对匹配文档进行排序。BM25 源自 20 世纪八九十年代伦敦城市大学第一个实



现这个函数的系统——Okapi 信息检索系统。它是基于 20 世纪七八十年代由 Stephen E. Robertson、Karen Spärck Jones 等人开发的概率检索框架。从 20 世纪 80 年代末开始，概率模型（特别是以 Okapi 系统为代表的 BM25 系列算法）出现并逐渐分享了经典模型在信息检索模型领域的地位，成为新兴的且功能强大、表现越来越出色的模型。

BM25 和 BM25F 两种模型在 TREC 文本检索评测会议中都有优越于其他的表现并且公认是目前 IR 范围内最为先进的检索模型。BM25 适用于没有结构的全文检索，而 BM25F 适用于结构化的文档检索，也就是用于有好几个全文搜索列的情况。

### 1. 排名函数

BM25 是一个基于单词集合的检索函数，它依据出现在每个文档中的查询词对匹配文档集合排序，而不管查询词在文档内相互之间的联系。它不是一个单一的函数，而是有略微不同的组件和参数变化的一群函数的集合。一个最典型的具体函数举例如下。

假定有一个查询词组  $Q$ ，含有关键词  $q_1, \dots, q_n$ ，用 BM25 给文档  $D$  评分的公式是：

$$SCORE(D, Q) = \sum_{i=1}^N IDF(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{D}{avgdl})}$$

其中  $f(q_i, D)$  是检索词  $q_i$  在文档  $D$  中的频率， $|D|$  是文档  $D$  以单词为单位的长度， $avgdl$  是抽取出的文档的文本集合的平均文档长度。 $k_1$  和  $b$  是自由参数，通常选择  $k_1 = 2.0$  和  $b = 0.75$ 。 $IDF(q_i)$  是检索词  $q_i$  的 IDF（文档频率倒数）权重。 $IDF(q_i)$  的通常计算公式是：

$$IDF(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

这里， $N$  是集合中文档的总数， $n(q_i)$  是包含  $q_i$  的文档个数。

对 IDF 的解释较多，例如有人用信息论来解释 IDF，所以使用 IDF 的公式也有一些变化。在最初的 BM25 发展过程中，IDF 组件来源于二元独立模型。

当把它应用于在一半以上的语料文档中出现的检索词（term）时，以上的 IDF 公式存在潜在的重大缺陷。这些检索词的 IDF 是负数，所以对于两个几乎相同的文件——一个包含检索词而另一个不包含它——后者可能得到更高的分数，这意味着检索词出现在一半以上的语料文档中时将对最后文档的相关性得分提供负贡献。通常我们并不期望这样，所以

许多实际的应用程序会以不同的方式处理 IDF 公式：

- 每个加数规定一个最小数 0，为了去除常见的检索词；
- 为了避免常见的检索词被完全忽略，给 IDF 方程一个常量  $\epsilon$ ；
- 为了避免检索词被完全忽略，IDF 方程被包含一些非负的数或者严格正数的类似形式方程代替。

## 2. IDF 的信息理论性解释

下面是一个来源于信息理论的解释。假设查询词  $q$  出现在  $n(q)$  个文档中，那么一份随机文档  $D$  包含查询词的概率为  $\frac{n(q)}{N}$ （这里的  $N$  指文本集合包含的文档总数）。因此，“ $D$  包含  $q$ ”的信息量是：

$$-\log \frac{n(q)}{N} = \log \frac{N}{n(q)}$$

现在假设我们有两个检索词  $q_1$  和  $q_2$ 。如果它们出现在完全不相关的文档中，那么随机文档  $D$  中  $q_1$  和  $q_2$  都出现的概率是： $\frac{n(q_1)}{N} \cdot \frac{n(q_2)}{N}$ ，相应的信息量是： $\sum_{i=1}^2 \log \frac{N}{n(q_i)}$ 。

加入一个小的变化，这恰好是 BM25 的 IDF 组件的公式表达。

Robertson 等人在简化 BM25 模型的基础上提出了词频加权的权重计算公式 BM25F。首先获得检索词在所有域上累计的权重：

$$weight(t, d) = \sum_c \sum_{in\ d} \frac{occurs_{t,s}^d \cdot boost_c}{\left( (1 - b_c) + b_c \cdot \frac{l_c}{avl_c} \right)}$$

$l_c$  是域的长度； $avl_c$  是域  $c$  的平均长度； $b_c$  是一个与域长度有关的常数， $b_c$  与 BM25 中的  $b$  相似； $boost_c$  是运用于域  $c$  的加权因子。然后，再用一个非线性饱和函数  $\frac{weight}{k_1 + weight}$ （饱和函数就是说输入达到一定的值以后输出就不再有大的变化了）：

$$R(q, d) = \sum_{t\ in\ q} IDF(t) \cdot \frac{weight(t, d)}{k_1 + weight(t, d)}$$



$$IDF(t) = \log \frac{N - df(t) + 0.5}{df(t) + 0.5}$$

这里的  $N$  是文档集中文档的个数， $df$  是出现过检索词  $t$  的文档的个数。

BM25F 的概率模型比较适合基于关键词来源位置进行加权的词频统计方法。它保证了词频的作用不会成为衡量关键词在文档中重要性的唯一标准；该模型综合考虑了词频、文档频、文档长度、文档集合平均长度等多种因素，尤其在 BM25F 模型中添加了赋予不同域的关键词不同权重的衡量参数，进一步保证了关键词的加权词频能够反映该关键词对文档主要内容的贡献程度。

Lucene 默认使用的检索模型是向量空间模型。为方便在 Lucene 中使用概率模型中著名的 BM25 模型必须定制 Lucene 的评分系统。

为了将 BM25 概率模型排序集成到 Lucene 需要开发一个新的 Query、Weight 和若干 Scorer。其主要功能是实施 Scorer 水平，因为 Query 和 Weight 的主要目的是为 Scorer 准备必要的参数及在调用搜索方法时创建 Scorer 实例。

在 Lucene 中，修改评分系统将比修改 similarity 能够更多地影响排序结果。Lucene 的评分系统是一个非常复杂的机制，主要由下面三个类来实现：

- Query——表示用户需要的信息的抽象类；
- Weight——用户 Query 的内部接口表示，以便能够重用 Query 对象；
- Scorer——包含评分公共功能的抽象类，提供了评分和解释评分结果的功能。

查询的执行可以分成两个部分：布尔过滤和排序评估。布尔过滤是通过依赖于逻辑运算符的 ShouldBooleanScorer、MustBooleanScorer 和 NotBooleanScorer 实现的；而排序评估则是通过 BM25TermScorer 和 BM25FTermScorer 的 score 方法来实现的。

依赖于调用的构造函数形式，BM25BooleanScorer 创建 BM25TermScorer 或 BM25FTermScorer 的实例，两个构造函数如下：

- `public BM25BooleanQuery(String query, String field, Analyzer analyzer) throws ParseException, IOException` 使用 BM25 排序函数；
- `public BM25BooleanQuery(String query, String[] fields, Analyzer analyzer) throws ParseException, IOException` 使用 BM25F 排序函数。

BM25BooleanScorer 将忽略任何 Lucene QueryParser 处理的和域有关的信息, 因此将只在构造函数中作为参数传入的域中执行搜索。另外, 它只支持布尔查询, 其他查询类型将被分割成检索词然后作为布尔查询执行。

应当指出的是, 这两个排名函数没有使用到查询权重, 因此, 所有的计算是在 scorer 水平上执行的。

几乎所有为了计算 BM25 相关的必要信息都能通过 Lucene expert API 获得, 例如 termdocs、numdocs、docfreq 等, 但文档的平均长度无法通过 Lucene 内部的 API 获得。可以在做索引的时候得到文档的平均长度, 可以在初始化的时候通过执行一个特定文件比较工具计算并存储文件长度来获得。取得文档平均长度的主要代码如下所示:

```
public class CollectionSimilarityIndexer extends DefaultSimilarity{
    private static Map<String,Long> length = new HashMap<String,
        Long>();

    @Override
    public float lengthNorm(String fieldName, int numTokens) {
        Long aux = CollectionSimilarityIndexer.length.
            get(fieldName);
        if (aux==null)
            aux = new Long(0);
        aux+=numTokens;
        CollectionSimilarityIndexer.length.put(fieldName,aux);
        return super.lengthNorm(fieldName, numTokens);
    }

    public static long getLength(String field){
        return CollectionSimilarityIndexer.length.get(field);
    }
}
```

在索引完成后, 可以检索到某个域的长度。把 CollectionSimilarityIndexer.length/numdocs 的值保存到一个文件中, 可以在打开搜索器的时候读取这个文件。BM25Parameters 的 load(String filePath)方法用来载入文档平均长度。

可以在 BM25Parameters 类中找到相关的 BM25 参数, 默认设置  $k_1=2$ ,  $b=0.75$ 。BM25F 的情况更为复杂, 因为 BM25F 需要更多的参数, 主要是一个包含被搜索域的字符串数组。可以在 BM25Fparameters 中找到所有的参数, 同样道理,  $k_1$  也是一个数组。对于  $b$  来说, 每个域均设置为 0.75。但是构建 Query 对象时, 可以使用更合适的参数来代替默认值。可



以用浮点数组来设置这些参数。`boost` 的情况有些相似，这些值已经初始化为 1，但却可以提供一个浮点型数组来修改初始值。必须有序提供所有基于 `BM25F` 的数组参数，例如 `boostfield` 和 `bfield`。这意味着对于域数组中的第  $i$  个域的参数 `boost` 和 `b` 在各自的数组中也处于第  $i$  个位置。

在两个模型中，`IDF` 是在 `BM25Similarity` 中计算的，并且必须用文档频率和 `numdocs` 这两个值在文档级进行计算。`Lucene` 在域级别返回文档频率，对于 `BM25` 非常合适，因为在 `BM25` 中只对唯一的域进行搜索。对于 `BM25F` 的情况却是一个问题，因为 `IDF` 只有在包含所有被索引的词的一个新的域中才能进行文档级别的计算，而提供的实现则在一个最长的平均长度的域中计算文档频率。

这个实现的用法和用 `Lucene` 搜索的方法类似，但是在查询执行前必须给 `BM25Parameters` 或 `BM25FParameters` 赋值，这样做是为了设置列的平均长度，而其他参数则可以忽略，因为它们被设置为默认值。使用 `BM25` 的例子如下所示：

```
IndexSearcher searcher = new IndexSearcher("Index//Path");
//加载平均长度
BM25Parameters.load(avgLengthPath);
BM25BooleanQuery query =
    new BM25BooleanQuery("This is my Query", "Search-Field",
        AnalyzerUtil.getPorterStemmerAnalyzer(new StandardAnalyzer()));

//取得归一化之后的评分值
Hits hits = searcher.search(query);

//打印结果
for (int i = 0; i < 10; i++)
    System.out.println(hits.id(i) + ":" + hits.score(i));
```

使用 `BM25F` 的例子如下所示：

```
String[] fields = {"FIELD1", "FIELD2"};
IndexSearcher searcher = new IndexSearcher("Index//Path");

//为每个列设置平均长度
BM25FParameters.setAverageLength("FIELD1", 123.5f);
BM25FParameters.setAverageLength("FIELD2", 42.2f);

//设置 k1 参数
BM25FParameters.setK1(1.2f);

//使用 boost 和 b 的默认值
```

```

BM25BooleanQuery queryF = new BM25BooleanQuery("This is my query",
    fields, AnalyzerUtil.getPorterStemmerAnalyzer(new Standard
    Analyzer()));

//得到不归一化的评分值
TopDocs top = searcher.search(queryF, null, 10);
ScoreDoc[] docs = top.scoreDocs;

//打印结果
for (int i = 0; i < top.scoreDocs.length; i++) {
    System.out.println(docs[i].doc + ":" + docs[i].score);
}

```

### 6.6.3 统计语言模型

语言模型最初来估计一段文本的生成概率，也可以用语言模型来估计一篇文档和某个查询词的相关程度。例如， $n$  元模型就是一种常用的语言模型。语言模型已经用在机器翻译、语音识别和手写体识别等自然语言处理相关的领域。

信息检索中的语言模型计算每个文档生成查询项的概率，基本公式是： $P(Q|M_d)$ 。

这里， $Q$  是查询字符串，其中包含查询词  $t_1, \dots, t_n$ 。  $M_d$  是文档  $d$  的语言模型。文档  $d_1, \dots, d_n$  可以根据  $P(Q|M_d)$  评分。

例如有 6 篇文档，计算文档生成查询项的概率如下：

$$P(Q|M_{d_1})=0.3; P(Q|M_{d_2})=0.1; P(Q|M_{d_3})=0.08;$$

$$P(Q|M_{d_4})=0.06; P(Q|M_{d_5})=0.18; P(Q|M_{d_6})=0.28$$

这 6 篇文档的排名如下：

Doc 1 0.3

Doc 6 0.28

Doc 5 0.18

Doc 2 0.1

Doc 3 0.08





Doc 4 0.06

在类似语音识别这样的应用中，使用了能够基于长序列预测单词的  $n$  元（ $n$ -gram）语言模型。 $n$  元模型基于前面的  $n-1$  个词来预测下一个词。最常用的  $n$  元模型是二元（bigram）模型和三元（trigram）模型。二元模型在前一个词的基础上预测下一个词，三元模型在前两个词的基础上预测下一个词。尽管二元模型已经用在信息检索中来表示两个词组成的短语，但是这里专门讨论一元模型，因为一元模型更简单并且作为搜索结果排序算法的基础很有效。

关于搜索的应用，我们使用语言模型表示文档的主题内容，关于信息检索的讨论中很少给主题一个定义。在这个方法中，我们把主题定义成在词汇上的概率分布，也就是语言模型。例如，一个文档是关于在嘉陵江钓鱼，我们会注意到在语言模型中和“钓鱼”与“位于嘉陵江”相关的词语有高概率。如果文档是一个关于在千岛湖钓鱼，有些高概率的词是一样的，但是会有更多的高概率词与“位于千岛湖”有关，如果这个文档是一个关于计算机的钓鱼游戏，很多高频词会与游戏制造和计算机使用相关，尽管这里仍然会有一些重要的词是关于钓鱼的。话题语言模型（简称话题模型）包含所有词的概率，不仅是最重要的词。大多数词有个默认概率，这个值在任何文档中都是一样的，但是对主题重要的词将有不同寻常的高概率。

可以把语言模型看成是朴素贝叶斯方法在文档检索领域的应用。把索引库中的  $n$  个文档都看成一个仅仅包含文档自身一个训练样本的类别，使用这个  $n$  类的分类器对查询分类，基于类别的后验概率对文档评分：

$$P(d | q) \propto \prod_{t \in q} (\lambda P(t | D) + (1 - \lambda)P(t))$$

这里使用了线性插值来平滑概率，其中：

$$P(t_i | D) = \lambda P(t_i | D) + (1 - \lambda)P(t_i)$$

这里的  $P(t_i | D)$  是从文档  $D$  生成查询词  $t_i$  的概率， $P(t_i)$  是从整个语料库生成查询词  $t_i$  的概率。

$\lambda$  值控制平滑量。 $\lambda$  的值越小，则越平滑，因为这时候查询词无条件的生成概率  $(1-\lambda)$  更大了。

设置合适的  $\lambda$  对于文档评分有重要的影响。可以根据一个查询项的集合人工或者自动设置最大性能。较低的  $\lambda$  对于长的查询更好，较高的  $\lambda$  对于短的查询更好。为了提高计算效率，不要计算不包括任何查询词的文档概率。



## 6.7 本章小结

本章介绍了 Lucene 全文索引库的基本使用方法和常用的定制修改方法，介绍了索引文件的格式，以及通过分发索引文件到其他服务器来实现分布式搜索。为了实现更好的搜索准确性，可以改进检索模型。在下一章中，我们将介绍索引库在用户界面中的调用方法。



## 第 7 章

# 搜索引擎用户界面

看搜索日志可能会想，搜索访问量比卖快餐的量差太多了。用户难得搜索一次，所以要尽量把返回页面的价值最大化。返回结果的信息可能是各种各样的，但一定都是用户想看到的信息，尽量不要列出任何无关的信息。现在各大搜索引擎的第一页基本都是综合页，同时返回新闻、图片或视频的搜索结果，都是聚合搜索（aggregated search），还有类似于 Naver (<http://www.naver.com>) 那样的综合页面。

对于互联网搜索来说，搜索结果界面往往采用 JSP 或 ASP.NET、PHP、Python 等技术来实现。搜索联想词的页面效果可以用 AJAX 来实现。为了实现更好的封装，可以结合 Spring 或 Struts 框架封装搜索请求，展现搜索结果。



### 7.1 实现 Lucene 搜索

在开发搜索 Web 界面之前，首先写一个控制台方式运行的搜索程序测试一下索引库：

```
Searcher searcher = new IndexSearcher(indexPath);
IndexReader reader = IndexReader.open(indexPath);
System.out.println("doc number in the index: " + reader.numDocs());
Query bodyQuery = null, titleQuery = null, query = null;
//对内容列的查询
QueryParser parser = new QueryParser("body", analyzer);
```

```

bodyQuery = parser.parse(queryString);
//对标题列的查询
parser = new QueryParser("title", analyzer);
titleQuery = parser.parse(queryString); //解析查询词
System.out.println("Searching for: " + bodyQuery.toString("body"));
BooleanQuery bodyOrTitle = new BooleanQuery();
bodyOrTitle.add(bodyQuery, BooleanClause.Occur.SHOULD);
bodyOrTitle.add(titleQuery, BooleanClause.Occur.SHOULD);
query=bodyOrTitle.rewrite(reader);
//需要这行语句来扩展搜索词
Hits hits = null;
//设置排序方式, 比如说高优先级的先显示
SortField classSortField = new SortField("class",SortField.INT,true);
Sort classSort = new Sort(new SortField[] {classSortField});
hits = searcher.search(query,classSort);
System.out.println("find place "+hits.length());
Highlighter highlighter =new Highlighter(new SimpleFormatter(),new
QueryScorer(query));
String text;
TokenStream tokenStream;
for (int i = 0; i< hits.length(); i++){
    text = hits.doc(i).get("body");
    //内容列的高亮显示
    TermPositionVector tpv = (TermPositionVector)
        reader.getTermFreqVector(hits.id(i),"body");
    tokenStream=TokenSources.getTokenStream(tpv,false);
    String result = highlighter.getBestFragment(tokenStream,text);
    System.out.println("body:"+result);
    //标题列的高亮显示
    text = TextHtml.text2html(hits.doc(i).get("title"));
    tokenStream=analyzer.tokenStream("title",new StringReader(text));
    result = highlighter.getBestFragment(tokenStream,text);
    if (result == null)
        System.out.println(hits.doc(i).get("title"));
    else
        System.out.println("title:"+result);
}
searcher.close();

```

查询对象 `Query` 的 `rewrite` 方法把复杂的查询条件重写成简单的查询条件。例如, `MultiTermQuery` 会扩展成很多的 `TermQuery`。`MultiTermQuery` 需要根据索引库中的值来扩展, 所以 `rewrite` 方法需要传入 `IndexReader` 参数。



## 7.2 实现搜索接口

本节介绍从基本的布尔逻辑查询开始，到指定范围的查询，以及搜索结果排序等实现方法。

### 7.2.1 编码识别

搜索引擎的查询关键词是很重要的一个参数，这个参数是一个查询字符串的 URL 编码。一个非 ASCII 字符的 URL 编码由一个“%”符号后面跟着两个十六进制的数字组成。中文搜索需要判断传入的这个字符串的 URL 编码是 GBK 还是 UTF-8 格式。

符合 J2EE 标准的 Web 服务器（例如 Tomcat）通过调用 `request.getQueryString()` 方法可以得到原始提交的参数。比如发送：

```
http://localhost/search.do?query=%B0%A1
```

`getQueryString` 方法得到的字符串是：

```
query=%B0%A1
```

然后调用编码识别方法，用正确的编码来解码。

```
String input = "%E6%B5%B7%E6%8A%A5%E7%BD%91";
String codingName=getEncoding(input);//判断编码
System.out.println(URLDecoder.decode(input, codingName));
//用正确的编码来解码
```

主要的开发工作是根据输入字符串判断编码。

GB2312 的字符编码范围在 %B0%A1~%F7%FE 之间，如表 7-1 所示。

汉字 Unicode 编码范围从 \u4e00 到 \u9fa5。UTF-8 汉字 URL 编码后的取值范围在：

```
%E4%B8%80~%E4%BF%BF
%E5%B8%80~%E5%BF%BF
%E6%B8%80~%E6%BF%BF
%E7%80%80~%E7%BF%BF
```

表 7-1 汉字编码对照表

字 符	编 码
啊	%B0%A1
阿	%B0%A2
鞍	%B0%B0
龢	%F7%FE

```
%E8%80%80~%E8%BF%BF
%E9%80%80~%E9%BE%A5
```

像左括号和右括号这样的 ASCII 编码小于 128 的字符编码都小于 %80，例如左括号字符编码是 %28，右括号字符编码是 %29，而所有的汉字编码，无论是 UTF-8 或 GBK，每个字节的编码都是大于或等于 %80。

```
//判断是否可能是 UTF8 编码的汉字
public static boolean isUtf8(String code1,String code2,String code3) {
    if (code1.compareTo("E4") >= 0 && code1.compareTo("E9") <= 0 &&
        code2.compareTo("80") >= 0 && code2.compareTo("BF") <= 0 &&
        code3.compareTo("80")>=0 &&code3.compareTo("BF")<=0) {
        return true;
    }
    return false;
}
//判断是否可能是 Gb2312 编码的汉字
public static boolean isGb2312(String code1,String code2) {
    if (code1.compareTo("B0") >= 0 && code1.compareTo("F7") <= 0 &&
        code2.compareTo("A0")>=0 &&code2.compareTo("FF")<=0) {
        return true;
    }
    return false;
}
//根据字符列表猜测字符编码
public static String getEncodeByList(List<String> code) {
    if(code.size() >= 2 && code.size()%2 == 1 && code.size()%3 == 0) {
        return "utf8";
    }
    else if(code.size() >= 2 && code.size()%2 == 0 && code.size()%3 != 0) {
        return "gbk";
    }
    else if(code.size()%6 == 0) {
        for(int m=0;m<code.size();m = m+6) {
            if( ! isUtf8(code.get(m), code.get(m+1), code.get(m+2)) &&
                isGbk(code.get(m), code.get(m+1)) &&
                isGbk(code.get(m+2), code.get(m+3)) ) {
                return "gbk";
            } else if(isUtf8(code.get(m), code.get(m+1), code.get(m+2)) &&
                ! isGbk(code.get(m), code.get(m+1)) ) {
                return "utf8";
            }
            if(! isUtf8(code.get(m+3), code.get(m+4), code.get(m+5)) &&
                isGbk(code.get(m+2), code.get(m+3)) &&
```



```
        isGbk(code.get(m+4), code.get(m+5)) ) {
            return "gbk";
        } else if(isUtf8(code.get(m+3), code.get(m+4), code.get(m+5)) &&
            !isGbk(code.get(m+2), code.get(m+3))) {
            return "utf8";
        }
    }
}
return "utf8";
}
```

根据有限状态机的思想把字符串切分成数组。首先定义状态类。

```
public enum CharType {
    Enter, //碰到%
    Code1, //碰到%后的第一个字符
    Code2, //碰到%后的第二个字符
}
```

然后根据上一个状态以及当前的字符决定下一个状态。进入下一个状态时，有可能执行判断字符编码的动作。

```
public static String getURLEncoding(String url) {
    List<String> codes = new ArrayList<String>();
    CharType currentSate = null;
    char c1='\0';
    char c2='\0';
    for(int i=0; i<url.length(); ++i) {
        char currentChar = url.charAt(i);
        if(currentChar == '%') {
            if(currentSate == CharType.Code2 ) {
                char[] s1 = {c1,c2};
                codes.add(new String(s1));
            }
            currentSate = CharType.Enter;
        }else if(currentSate==CharType.Enter) {
            c1 = currentChar;
            currentSate = CharType.Code1;
        }else if(currentSate==CharType.Code1) {
            c2 = currentChar;
            currentSate = CharType.Code2;
        }else if(currentSate==CharType.Code2) {
            char[] s1 = {c1,c2};
            codes.add(new String(s1));
        }
    }
}
```

```

        currentSate = null;
        return getEncodeByList(codes);
    }
}
if(currentSate==CharType.Code2) {
    char[] s1 = {c1,c2};
    codes.add(new String(s1));
}
return getEncodeByList(codes);
}

```

## 7.2.2 布尔搜索

用布尔查询来实现多个查询条件的合并，最常见的例子是搜索标题或正文。

```

BooleanQuery bodyOrTitle = new BooleanQuery();
bodyOrTitle.add(bodyQuery, BooleanClause.Occur.SHOULD);
bodyOrTitle.add(titleQuery, BooleanClause.Occur.SHOULD);

```

这里 `BooleanClause.Occur.SHOULD` 代表“或者”的关系，如果要“并且”就用 `BooleanClause.Occur.MUST`。

也可以使用 `MultiFieldQueryParser` 来合并对多个列的搜索，比如下面实现对“body”和“title”两列的查找。

```

Query query =
    MultiFieldQueryParser.Parse(queryWord, new string[]
    {"body","title" }, analyzer);

```

## 7.2.3 指定范围搜索

在商品搜索中，经常需要指定按时间或价格等数值条件查找，如图 7-1 所示。

可以通过 `RangeQuery` 来实现这样的时间条件区间查找：

```

java.util.Calendar upper = GregorianCalendar.
getInstance();
upper.add(java.util.Calendar.YEAR, +100);
String t2 = formatter.format(upper.getTime());

if ("1".equals(dateRange)) {
    //一周内

```

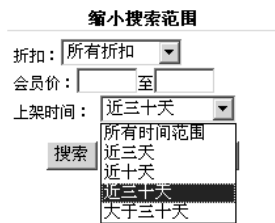


图 7-1 指定范围搜索实例





```
now.add(java.util.Calendar.DATE, -7);
String t1 = formatter.format(now.getTime());
ConstantScoreRangeQuery dateQuery =
    new ConstantScoreRangeQuery("time", t1, t2, true, true);
}
else if ("2".equals(dateRange)){
    //一月内
    now.add(java.util.Calendar.MONTH, -1);
    String t1 = formatter.format(now.getTime());
    ConstantScoreRangeQuery dateQuery =
        new ConstantScoreRangeQuery("time", t1, t2, true, true);
}
else if ("3".equals(dateRange)){
    //三月内
    now.add(java.util.Calendar.MONTH, -3);
    String t1 = formatter.format(now.getTime());
    ConstantScoreRangeQuery dateQuery =
        new ConstantScoreRangeQuery("time", t1, t2, true, true);
}
else if ("4".equals(dateRange)){
    //六月内
    now.add(java.util.Calendar.MONTH, -6);
    String t1 = formatter.format(now.getTime());
    ConstantScoreRangeQuery dateQuery =
        new ConstantScoreRangeQuery("time", t1, t2, true, true);
}
```

如果是在 Solr 界面中，区间条件的查询语法例子如下：

```
+汽车 +expiretime:[2007-08-13T00:00:00Z TO 2008-08-13T00:00:00Z]
```

如果要搜索单个日期值，需要对“:”转义，例如：

```
postdate:2007-08-13T00\:00\:00Z
```

## 7.2.4 搜索结果排序

可以按单列或者多列排序，但是需要保证排序列是不做切分处理的，也就是对该列做索引的时候设置 `Field.Index.NOT_ANALYZED`。例如“url”网址列没有做过切分，可以按该列排序，而标题列“title”做过切分，不能按该列排序。

经常需要按日期倒排序，为了支持对日期列排序，需要把日期转换成统一的字符串格式“yyyyMMddHHmmssSSS”。如果精度低，字符串长度相应变短。

索引日期的例子如下所示：

```
Date pubDate = rs.getDate("pubDate");
Field f = new Field("pubDate",
    DateTools.dateToString(pubDate, DateTools.Resolution.DAY),
    //精度到天
    Field.Store.YES,
    Field.Index.NOT_ANALYZED);
```

按日期倒排序的例子如下所示：

```
Sort sort= new Sort(new SortField("pubDate",SortField.STRING,true));
ScoreDoc[] hits = searcher.search(query,null,1000,sort).scoreDocs;
```

也可以对多个字段排序，比如先按地区列“area”排序，然后按类别“type”排序：

```
Sort sort= new Sort(new SortField[]{new SortField("area"),new SortField
("type")});
ScoreDoc[] hits = searcher.search(query,null,1000,sort).scoreDocs;
```

也可以通过 `SortComparatorSource` 自定义排序方法。

### 7.2.5 搜索页面的索引缓存与更新

索引一般是一个比较大的文件，一般从几百 MB 到几个 GB 不等。页面执行搜索的时候打开大的索引往往是一个非常耗时的过程。一般情况下需要缓存 `IndexReader` 和 `Searcher`，不要每次响应用户的搜索请求都重新打开索引后再执行搜索。

```
private static IndexReader reader = null;
private static Searcher searcher = null;
public void init(String indexPath) throws Exception {
    if(searcher != null)
        searcher.close();
    if(reader != null)
        reader.close();
    _dir = indexPath;
    searcher = new IndexSearcher(indexPath);
    reader = IndexReader.open(indexPath);
}
```

因为后台在更新，前台的缓存会导致索引更新不及时，不能搜到已经更新的内容，这时候就需要重新装载索引库。下面的代码实现过一段时间就检查索引的版本号，如果有更新则重新加载索引库。



```
private void refreshIndexReader() {
    //如果检查时间已经到了，就检查当前索引的版本号
    if ((LastCheckTime+interval) < System.currentTimeMillis()) {
        long newIndexVersion;
        newIndexVersion = IndexReader.getCurrentVersion(_dir);
        //如果索引已经是最新的，就重新设置检查时间
        if (newIndexVersion == currentIndexVersion) {
            LastCheckTime = System.currentTimeMillis();
            return;
        }

        synchronized (this) { //同步
            LastCheckTime = System.currentTimeMillis();
            searcher.close();
            reader.close(); //旧的 IndexReader 有可能还在被其他线程使用着
            reader = IndexReader.open(_dir);
            searcher = new IndexSearcher(reader);
            currentIndexVersion = newIndexVersion;
        }
    }
}
```

需要防止关闭还在被其他线程使用着的 `IndexReader`。`IndexSearcher` 不会增加对 `IndexReader` 的引用，也就是说不会调用 `IndexReader` 的 `incRef` 方法。`IndexReader` 的 `close` 方法只是调用 `decRef` 方法，减少对 `IndexReader` 实例的引用。如果引用计数降低到 0，则会实际关闭 `reader`。设计一个 `SearcherManager` 来管理 `IndexReader` 的缓存。

```
public class SearcherManager {
    private IndexSearcher currentSearcher; // 当前的 IndexSearcher
    private Directory dir;

    public SearcherManager(Directory dir) throws IOException {
        this.dir = dir;
        //创建初始的 IndexSearcher
        currentSearcher = new IndexSearcher(IndexReader.open(dir));
    }

    public void warm(IndexSearcher searcher) {
    } //预热新的 IndexSearcher

    private boolean reopening;

    private synchronized void startReopen() throws InterruptedException{
        while (reopening) {
```

```

        wait();
    }
    reopening = true;
}

private synchronized void doneReopen() {
    reopening = false;
    notifyAll();
}

//重新打开 IndexSearcher
public void maybeReopen() throws InterruptedException, IOException{
    startReopen();
    try {
        final IndexSearcher searcher = get();
        try {
            long currentVersion = currentSearcher.getIndexReader().
                getVersion();
            if (IndexReader.getCurrentVersion(dir) != currentVersion){
                IndexReader newReader = currentSearcher.getIndex
                    Reader()
                        .reopen();
                assert newReader != currentSearcher.getIndex
                    Reader();
                IndexSearcher newSearcher = new IndexSearcher
                    (newReader);
                warm(newSearcher);
                swapSearcher(newSearcher);
            }
        } finally {
            release(searcher);
        }
    } finally {
        doneReopen();
    }
}

public synchronized IndexSearcher get() { //返回当前的 IndexSearcher
    currentSearcher.getIndexReader().incRef();
    return currentSearcher;
}

public synchronized void release(IndexSearcher searcher)
//释放 IndexSearcher
    throws IOException {

```



```

        searcher.getIndexReader().decRef();
    }

    private synchronized void swapSearcher(IndexSearcher newSearcher)
        throws IOException {
        release(currentSearcher);
        currentSearcher = newSearcher;
    }
}

```

在 Web 服务器中使用 SearcherManager 类的方法如下面的代码所示：

```

//多个线程之间共享同一个 SearcherManager
static SearcherManager searcherManager = new SearcherManager(indexdir);

//在每次搜索时调用如下方法
searcherManager.maybeReopen();
IndexSearcher searcher = searcherManager.get();
try {
    //执行搜索和结果输出
} finally {
    searcherManager.release(searcher);
}

```



## 7.3 历史搜索词记录

可以通过 Cookie 记录用户经常搜索的关键字，然后就可以从用户经常搜索的关键字来判断用户的兴趣。先看一下怎么设置用户查询词。Cookie 在用户电脑中是以一种类似 map 的方式存放，且只能存放字符串类型的对象。通过 response 对象增加 Cookie，代码如下所示：

```

Cookie cookie = new Cookie("query", query);
cookie.setMaxAge(60*60*24*30); //设置 cookie 的存放时间(单位是秒)。
//然后通过 response 对象的 addCookie 方法添加 cookie 才能生效。
response.addCookie(cookie);

```

通过 request 对象的 getCookies 方法得到一个包含所有 Cookie 的数组。

```

Cookie[] cookies = request.getCookies();
//然后遍历这个数组就能得到记录查询词的 cookie
String query = null;
for (int i = 0; i < cookies.length; i++) {

```

```

        Cookie c = cookies[i];
        if (c.getName().equals("query")) {
            query = c.getValue();
        }
    }
}

```

上面的例子显示了如何设置并获取名称叫做 query 的 Cookie。如果要记录 5 个查询词，则需要设置 5 个不同的 Cookie。如果要在 Web 界面显示历史查询词，则需要把这些关键词去重，然后再显示出来。



## 7.4 实现关键词高亮显示

在搜索结果中一般都有和用户搜索关键词相关的摘要。关键词一般都会高亮显示出来。从实现上说就是把要突出显示的关键词前加上 “<B>” 标签，关键词后加上 “</B>” 标签。Lucene 的 highlighter 包可以做到这一点。

```

doSearching("汽车");
//使用一个查询初始化 Highlighter 对象
Highlighter highlighter = new Highlighter(new QueryScorer(query));
//设置分段显示的文本长度
highlighter.setTextFragmenter(new SimpleFragmenter(40));
//设置最多显示的段落数量
int maxNumFragmentsRequired = 2;
for (int i = 0; i < hits.length(); i++) {
    //取得索引库中存储的原始文本
    String text = hits.doc(i).get(FIELD_NAME);
    TokenStream tokenStream=analyzer.tokenStream(FIELD_NAME,
                                                    new StringReader(text));

    //取得关键词加亮后的结果
    String result = highlighter.getBestFragments(tokenStream,
                                                  text,
                                                  maxNumFragmentsRequired,
                                                  "...");

    System.out.println("\t" + result);
}

```

QueryScorer 设置查询的 query，这里还可以加上对字段列的限制，比如只对 body 条件的 Term 高亮显示，可以使用：new QueryScorer(query,"body")。



为了实现关键词高亮，必须知道关键词在文本中的位置。对英文来说，可以在搜索的时候实时切分出位置。但是中文分词的速度一般相对来说慢很多。在 Lucene1.4.3 以后的版本中，Term Vector 支持保存 Token.getPositionIncrement() 和 Token.startOffset() 以及 Token.endOffset() 信息。利用 Lucene 中新增加的 Token 信息的保存结果以后，就不需要为了高亮显示而在运行时解析每篇文档。为了实现一系列的高亮显示，索引的时候通过 Field 对象保存该位置信息。

```
//增加文档时保存 Term 位置信息
private void addDoc(IndexWriter writer, String text) throws IOException{
    Document d = new Document();

    Field f = new Field(FIELD_NAME, text ,
                        Field.Store.YES, Field.Index.TOKENIZED,
                        Field.TermVector.WITH_POSITIONS_OFFSETS);

    d.add(f);
    writer.addDocument(d);
}
//利用 Term 位置信息节省 Highlight 时间
void doStandardHighlights() throws Exception{
    Highlighter highlighter =new Highlighter(this,new QueryScorer(query));
    highlighter.setTextFragmenter(new SimpleFragmenter(20));
    for (int i = 0; i < hits.length(); i++) {
        String text = hits.doc(i).get(FIELD_NAME);
        int maxNumFragmentsRequired = 2;
        String fragmentSeparator = "...";
        TermPositionVector tpv =
            (TermPositionVector) reader.getTermFreqVector(hits.id(i),
                FIELD_NAME);
        TokenStream tokenStream=TokenSources.getTokenStream(tpv);

        String result = highlighter.getBestFragments(
            tokenStream,
            text,
            maxNumFragmentsRequired,
            fragmentSeparator);

        System.out.println("\t" + result);
    }
}
```

最后把 highlight 包中的一个额外的判断去掉。对于中文来说没有明显的单词界限，所以下面这个判断是错误的：

```
tokenGroup.isDistinct(token)
```

注意上面的 `highlighter.setTextFragmenter(new SimpleFragmenter(20));` 这句话。`SimpleFragmenter` 是一个最简单的段落分割器。它把文章分成 20 个字的一个段落。这种方式简单易行，但显得比较初步。有时候会有一些没意义的符号出现在摘要的起始部分，如图 7-2 所示。

**营销宣传策划：企业如何借媒体之力打开市场**  
，无不如群雄逐鹿中原。初是跨国公司如宝洁、联合利华进军中国市场发动兼并，欧莱雅收购著名品牌小护士等形成合纵之势。而后是我国各汽车企业开始大规模的整合，第一汽车集团和天津汽车集团重组，一汽控股天汽的优良资产——夏利

`RegexFragmenter` 是一个改进版本的段落分割器。它通过一个正则表达式匹配可能的热点区域。但它是为英文定制的。我们可以让它认识中文的字符段。

图 7-2 摘要之前符号实例

```
protected static final Pattern textRE = Pattern.compile("[\\w\\u4e00-\\u9fa5]+");
```

这样使用 `highlighter` 就变成了：

```
highlighter.setTextFragmenter(new RegexFragmenter(descLenth));
```

## 7.5 实现分类统计视图

一个职位搜索网站需要统计出某一关键词下要求本科学历的有多少岗位，要求专科学历的有多少岗位，薪资范围在 4000~6000 元的有多少岗位，薪资范围在 6000~8000 元的有多少岗位。从术语上讲，就是要从各个角度（维）进行分类并统计搜索结果数在相关分类中的分布情况，如图 7-3 所示。这个功能叫做搜索结果分类统计（Faceted search）。分类可以是多层次的，用户可以沿着某一类继续细化，这有点像数据仓库中的向下钻取，但它不是用数据库而是用 Lucene 完成的。这也是 Lucene 的一个很有特色的应用案例。



图 7-3 搜索结果分类显示实例

按指定类别搜索，对搜索结果的分类统计功能，这两个功能是不一样的，但用户开始并没有意识到。按类别搜索类似 SQL 语句的 `where` 条件，分类统计类似 SQL 语句中的 `Group by` 功能。

可以利用 `QueryFilter` 来实现搜索结果分类统计。



QueryFilter 有个 bits 方法返回一个 BitSet 集，这个 BitSet 的大小是所针对的 Lucene 库的大小（也就是 new BitSet(reader.max Doc())），凡符合 filter 条件的文档在位集合中相应位置上置为 1（true）。这个 BitSet 集对特定 QueryFilter 对象来说是 cache 保存的，下次调用不会重新计算。然后利用这个 BitSet 集，将满足各个基本属性值的 BitSet 值计算出，根据特定用户需要进行相关的 BitSet 与（交集）操作，最后利用 BitSet 集的 cardinality() 方法就可计算出满足该类的总数。计算流程如图 7-4 所示。



图 7-4 分类统计计算流程

用 QueryFilter 实现搜索结果分类统计的参考代码如下所示：

```
String[] cats = {"001004003","001008003021","001004014" }; //类别数组
long[] catCounts = new long[cats.length]; //分类统计结果

//原始查询
Filter all = new QueryWrapperFilter(q);

//用 AND 逻辑合并 Filter
ChainedFilter.DEFAULT = ChainedFilter.AND;
for (int i=0;i<cats.length;++i) {
    //分类统计查询条件
    Filter these = new QueryWrapperFilter(new TermQuery(new Term("cat",
        cats[i])));
    ChainedFilter chainedFilter = new ChainedFilter(
        new Filter[]{all,these}
    );
    //计算 Filter 中的 BitSet 的 1 的个数
    catCounts[i] = chainedFilter.getCardinality(reader);
}
return catCounts;
```

在 Apache 的另外一个企业搜索项目 Solr 中，通过优化后的 BitSet 实现了一个 DocSetHitCollector 来做分组求和。这个优化后的 BitSet 叫做 OpenBitSet。但是这个 OpenBitSet 只是在 64 位的机器上，和当返回的结果数量很多的时候才比 Java 内部的 BitSet 类更快。

计算一个二进制的数组的 1 的个数（叫做 PopCount 或者 cardinality）在整个计算中对

性能有比较重要的影响。这里把这个功能叫做计算数组的二进制势。通过查找表 7-2 可以快速地计算从 0 到 255 的二进制势。

表 7-2 计算从 0 到 255 数值的二进制数据中 1 的出现次数

数 值	0	1	2	3	4	...	255
二进制形式	00000000	00000001	00000010	00000011	00000100	...	11111111
1 的个数	0	1	1	2	1	...	8

下面是查表法实现的计算数组二进制势的程序代码：

```
private static int[] _bitsSetArray65536 = null;
static {
    _bitsSetArray65536 = new int[65536]; //16 位整数的二进制势表
    byte[] _bitsSetArray256 = { 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2,
        3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 1,
        2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4,
        5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3,
        4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3,
        4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4,
        5, 5, 6, 5, 6, 6, 7, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4,
        5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3,
        4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5,
        6, 5, 6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3,
        4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 3, 4, 4, 5, 4, 5, 5,
        6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6,
        7, 7, 8 }; //8 位整数的二进制势表
    //根据 8 位整数的二进制势表生成 16 位整数的二进制势表
    for (int j = 0; j < 65536; j++) {
        _bitsSetArray65536[j] = _bitsSetArray256[j & 0xff]
            + _bitsSetArray256[(j>> 8 )& 0xff];
    }
}

//计算给定数组 A 的二进制势表
public static long pop_array2(long A[],int wlen) {
    long _count = 0;
    for (int i = 0; i < wlen; i++) {
        _count += _bitsSetArray65536[(int) (A[i]& 0xffff)]
            + _bitsSetArray65536[(int) ((A[i] >> 16 )& 0xffff)]
            + _bitsSetArray65536[(int) ((A[i] >> 32) & 0xffff)]
            + _bitsSetArray65536[(int) ((A[i] >> 48) & 0xffff)];
    }
    return _count;
}
```

查表法和 Java 内部实现同样功能的 bitCount 方法测试比较性能：

```
long x = 1000000000000000001;
for(int i=0;i<1000;i++){
    pop(x); //查表法实现的位计算
}
long end = System.nanoTime();
System.out.println(end-start); //输出计算时间

long start2 = System.nanoTime();
for(int i=0;i<1000;i++){
    Long.bitCount(x); //java 内部实现的实现的位计算
}
long end2 = System.nanoTime();
System.out.println(end2-start2); //输出计算时间
```

下面的搜索结果分类统计功能实现通过 DocSet 的 intersectionSize 方法减少计算步骤，又比上面的实现快了一些。

```
System.String[] mfgs = new System.String[] { "105", "93", "125" };
//类别数组

DocSetHitCollector all = new DocSetHitCollector(reader.MaxDoc());
searcher.Search(q, all);
DocSet allDocSet = all.DocSet;

int[] mfg_counts = new int[mfgs.Length];

for (int i = 0; i < mfgs.Length; ++i){
    DocSetHitCollector these = new DocSetHitCollector(reader.MaxDoc());
    searcher.Search(new TermQuery(new Term("type", mfgs)), these);
    //集合求交运算和计算集合大小运算两个操作
    //在 QueryFilter 方法中是分开计算和独立优化的，
    //现在把这两个操作放入一个函数中整体优化来提高程序运算效率。
    mfg_counts = these.DocSet.intersectionSize(allDocSet);
}
```

上面这个实现比起最初的 QueryFilter 实现，在于合并了以下两个步骤：

```
these.and(all);
mfg_counts[i] = these.cardinality();
```

二级子树展开的效果图如图 7-5 所示。

ipod nano (Best Matching categories for your search)			
<b>Consumer Electronics</b> MP4 Players & Access.. (32780) ⊕ <a href="#">Show All</a>	<b>MP4 Players &amp; Access..</b> MP4 Players (30494) MP4 Accessories (2208)	<b>MP3 Players &amp; Access..</b> MP3 Players (6869) MP3 Accessories (807)	<b>Ipod Accessories</b> iPod Cases (631) other iPod accessori.. (318) ⊕ <a href="#">Show All</a>

图 7-5 二级子树的展开效果

二级子树展开的实现代码如下所示：

```
//搜索形成的二级子树
HashMap<Integer, CountNode> cat1Set = new HashMap<Integer, CountNode>();
for (Count c : facetCounts) {
    Integer cat2Id = Integer.parseInt(c.getName());
    CatNode cat2Node = catMap.get(cat2Id);
    CountNode newParen = cat1Set.get(cat2Node.parent.no);

    if (limitCat > 0) {
        if (cat2Node.parent.no != limitCat) {
            continue;
        }
    }

    if (newParen == null) {
        newParen = new CountNode(cat2Node.parent.no,
            cat2Node.parent.name, null, false);
        CountNode childNode = new CountNode(cat2Node.no, cat2Node.name,
            newParen, true);
        childNode.count = c.getCount();
        newParen.children.add(childNode);
        cat1Set.put(newParen.no, newParen);
    } else {
        CountNode childNode = new CountNode(cat2Node.no, cat2Node.name,
            newParen, true);
        childNode.count = c.getCount();
        newParen.children.add(childNode);
    }
}
```

使用 MatchAllDocsQuery 返回所有的记录，然后再分类统计，这样可以实现一个分类导航的页面。



## 7.6 实现 Ajax 搜索联想词

搜索输入框中的下拉提示给用户一个有参考意义的搜索词表，同时也提供用户搜索该词预期的结果数量，如图 7-6 所示。这个功能一般是由浏览器端的 Ajax 代码完成的。

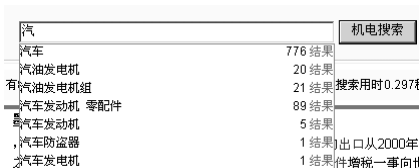


图 7-6 搜索下拉框提示效果图

搜索词表可以从用户搜索日志中统计出来，搜索次数多的词排在前面。除了按一般的设计，为每个用户提供统一的词表，还可以对每个用户提供个性化的推荐词表。例如：用户输入“汽”时，“汽车”的搜索次数比“汽油发电机”多，所以排在提示词列表的前面。如果搜索日志比较少，无法挖掘出足够多的推

荐搜索词，可以考虑从文本中挖掘一些关键词作为推荐搜索词。因为后台索引一般都在不断地变化，推荐搜索词右侧显示的“\*\*结果”并不是实时搜索出的结果，只是一个估计值，只具有参考价值。为了实现搜索提示效果，需要用到词典的前缀匹配。

### 7.6.1 估计查询词的文档频率

为了对用户输入任何词都可以显示一个估计的搜索结果数量，需要计算这个词的文档频率。例如用户输入“NBA 直播”，可以根据“NBA”和“直播”的文档频率估计“NBA 直播”的文档频率。假设“NBA”和“直播”之间的出现没有依赖关系，则可以简化计算如下：

$$P("NBA" | "直播") = P("NBA")gP("直播")$$

这里的  $P("NBA" | "直播")$  是联合概率，可以认为是“NBA 直播”的出现概率，而  $P("NBA")$  和  $P("直播")$  是每个词出现的概率。假设索引中的总文档数量是  $N$ ，而  $P("NBA") = \text{Freq}("NBA")/N$ ， $P("直播") = \text{Freq}("直播")/N$ 。

因此“NBA 直播”的文档频率

$$\text{Freq}("NBA直播") = N \times \frac{\text{Freq}("NBA")}{N} \times \frac{\text{Freq}("直播")}{N}$$

而  $\text{Freq}("NBA")$  和  $\text{Freq}("直播")$  所代表的文档频率在 Lucene 中可以通过 `org.apache.lucene.search.Searcher` 的 `docFreq(Term term)` 方法得到。因为多个词之间并不一

定满足独立出现的假设，因此这个估计值有可能偏低。

### 7.6.2 搜索联想词总体结构

当用户在浏览器的输入框输入查询词时，JavaScript 代码捕获用户即时输入的数据并向服务器发送请求。自动完成功能的总体结构如图 7-7 所示。

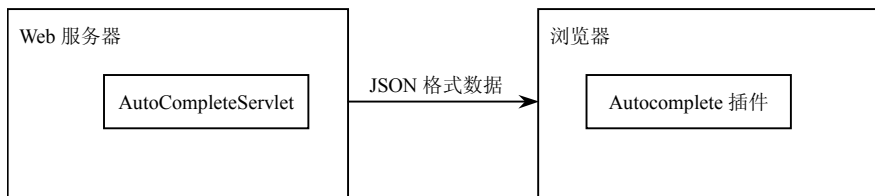


图 7-7 自动完成功能总体结构

### 7.6.3 服务器端处理

当用户输入一个搜索字的同时由 Web 服务器中的一个 Servlet(AutoCompleteServlet)从后台取数。

```

CREATE TABLE [keywordAnalysis] (
    [SearchTerms] [varchar] (50) NOT NULL ,    --搜索词
    [AccessCount] [int] NULL ,                  --用户搜索词数
    [Result] [int] NULL ,                       --搜索返回结果数
    [Count] [varchar] (50) NULL                 --搜索返回结果数的字符型表示
)
  
```

AutoCompleteServlet 的主要代码如下所示：

```

StringBuilder message = new StringBuilder("<ul>");

drv = (Driver)Class.forName(driver).newInstance();
DriverManager.registerDriver(drv);
con = DriverManager.getConnection(url,user,password);

String sql = "SELECT top 9 [searchTerms] , [Count] FROM [KeywordAnalysis]
WHERE ([searchTerms] LIKE '"+val+"%') and [searchTerms]!='"+val+"' and
result>0";
PreparedStatement stmt = con.prepareStatement(sql);
ResultSet rs = stmt.executeQuery();

while (rs.next()){
    String Word = rs.getObject(1).toString();
  
```



```
String Count = rs.getObject(2).toString();

    message.append("<li style=\"font-size:12px;padding:0px;height:15px;line-height:15px;overflow:hidden\"><div style=\"text-align:left;float:left\">");
    message.append(Word);
    message.append("</div><div style=\"text-align:right;float:right\"><span class=\"informal\">");
    message.append(Count);
    message.append("&nbsp;  <font color=\"#009900\">结果</font></span></div></li>");
}
rs.close();
con.close();
stmt.close();

message.append("</ul>");

response.setContentType("text/html; charset=utf-8");
response.setCharacterEncoding("utf-8");
PrintWriter out = response.getWriter();
out.println(message.toString());
```

把 `AutoCompleteServlet` 通过 `web.xml` 部署到 URL 地址 “`/autoComplete`”。

```
<servlet>
    <servlet-name>AutoCompleteServlet</servlet-name><servlet-class>com.lietu.autocomplete.AutoCompleteServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>AutoCompleteServlet</servlet-name>
    <url-pattern>/autoComplete</url-pattern>
</servlet-mapping>
```

#### 7.6.4 浏览器端处理

剩下的就是在前台通过 Ajax 组件库 jQuery 中的 `Autocomplete` 插件 (<http://docs.jquery.com/Plugins/autocomplete>) 来完成显示了。用户选择词后，一般直接跳转到这个词的搜索结果，而不是只在输入框显示这个搜索词，之后用户需要再次按搜索按钮才返回搜索结果。

部署总体流程说明如下。

- ① 先到官方网站下载 jQuery 的最新版本和 `AutoComplete` 插件。

② 将插件中的 JavaScript 文件和 css 文件分别置于 Web 项目中的 js 文件夹和 css 文件夹中, 例如 js/jquery-1.4.2.js 和 js/jquery.autocomplete.js 文件, 以及 jquery.autocomplete.css 文件。

③ 将这些文件导入到需要搜索联想词的页面, 一般是搜索首页和搜索结果页面, 也就是网页的头信息中包含这些文件。

```
<head>
<link rel="stylesheet" type="text/css" href="./css/jquery.
autocomplete.css"/>
<script language="javascript" src="js/jquery-1.4.2.js"></script>
<script language="javascript" src="js/jquery.autocomplete.js">
</script>
</head>
```

AutoComplete 插件与一个 HTML 的 Input 标签相结合。

为了方便跟踪错误, 可以使用 FireFox 中的 Firebug 插件调试网页中的 JavaScript 代码。在 Firebug 中, 可以为 JavaScript 设置断点, 可以暂停执行 JavaScript 并且看到每个变量的当前值。如果代码速度慢, 还可以通过 JavaScript 配置器查看性能, 快速发现性能瓶颈。

服务器传递给客户端的 JSON 数据格式由两列组成: w 表示搜索词, c 表示搜索次数, 例如:

```
var searchs = [
    { w: "北京", c: "1000" },
    { w: "iPod", c: "12" },
    { w: "iPhone", c: "3" },
    { w: "北方", c: "6" }
];
```

为了达到在下拉列表中显示如图 7-8 所示的两列效果, 修改 jquery.autocomplete.js 中的 function fillList():

```
var li = $("<li/>").html( options.highlight(formatted, term) ).addClass
(i%2 == 0 ? "ac_even" : "ac_odd").appendTo(list)[0];
```

将其改成如下形式:

```
var li = $("<li style=\"font-size:12px;padding:0px;height:15px;line-
```



图 7-8 下拉列表中显示两列的效果





```
height:15px;overflow:hidden\ "><div style=\ "text-align:left;float:left\
">"+data[i].data.w+"</div><div style=\ "text-align:right;float:right\ ">
<font color=\ "#009900\ ">"+data[i].data.c+"</font></div></li>").appendTo
(list)[0];
```

为了能解析 JSON 格式的返回结果，修改 jquery.autocomplete.js 中的 parse:

```
function parse(data) {
    var parsed = [];
    var rows = eval('(' + data + ')');
    if(rows) {
        for (var i=0; i < rows.length; i++) {
            var row = rows[i];
            if (row) {
                parsed[parsed.length] = {
                    data: row,
                    value: row.w,
                    result: row.w
                };
            }
        }
    }
    return parsed;
};
```

Web 界面部分代码首先增加一个 id="query" 的输入框到搜索页面:

```
<input autocomplete="off" type="text" name="query" id="query" class=
"ipt" value="请输入查询目的地..." onclick="this.value=''"/>
```

然后这个搜索页面调用修改后 jQuery 的 autocomplete 插件:

```
<script language="javascript" type="text/javascript">
$.ready(function()
{
    //设置成 POST 方式发送数据给 Servlet
    $.ajaxSetup({type: 'POST'}); //用于设置以 post 方式向后台提交
    //将 Autocomplete 控件与 id 为 query 的 input 控件相绑定并设置自动提示效果
    $("#query").autocomplete("./autoComplete",
    {
        minChars: 0,
        width: 360,
        matchContains: true,
        autoFill: false,
```

```

        formatItem: function(row, i, max)
        {
            return i + "/" + max;
        },
        formatMatch: function(row, i, max)
        {
            return row.w;
        },
        formatResult: function(row)
        {
            return row.w;
        }
    });
});
</script>

```

有时候需要支持用户选择提示词直接跳转到搜索结果页面。在 JavaScript 中，通过 `location.href` 实现跳转，例如：

```

function searchNow(){
    location.href="searchResult.jsp";
}

```

当用户选择一个值后，会执行 `result(handler)` 中指定的 `handler`，可以在这里跳转到搜索结果页。

```

$("#query").autocomplete(data, {
    formatItem: function(item) {
        return item.text;
    }
}).result(function(event, item) {
    location.href = "searchResult.jsp?query="+w;
});

```

服务器端修改成如下形式：

```

public void doPost(HttpServletRequest request,
                    HttpServletResponse response)
    throws IOException {
    String val = request.getParameter("q");
    String message = null;
    SuggestItem[] items = null;
    try {
        SuggestDic sugDic = SuggestDic.getInstance();

```



```
        items = sugDic.matchPrefix(val, 10); //查找 trie 树

        JSONValue lMyValue = JSONMapper.toJSON(items);
        message = Escape.toUnicodeEscapeString( lMyValue.render
            (false) );

    } catch (Exception e) {
        e.printStackTrace();
    }

    response.setContentType("text/html; charset=utf-8");
    PrintWriter out = response.getWriter();
    if(message!=null)    {
        out.println(message);
    } else {
        out.println("");
    }
}
```

可以通过 EasyMock 测试这个 Servlet 的返回值：

```
String queryWord = "P";
//录制 mock 对象
HttpServletRequest request = createMock(HttpServletRequest.class);
HttpServletResponse response = createMock(HttpServletResponse.class);
ServletConfig servletConfig = createMock(ServletConfig.class);
ServletContext servletContext = createMock(ServletContext.class);

AutoCompleteServlet instance = new AutoCompleteServlet();

//初始化 servlet, 一般由容器承担, 用 servletConfig 作为参数初始化, 此处模拟容器行为
instance.init(servletConfig);
//在某些方法被调用时设置期望的返回值,
//如下这样就不会去实际调用 servletConfig 的 getServletContext 方法, 而是直接返回
//servletContext, 由于 servletConfig 是 mock 出来的, 所以可以完全控制。
expect(servletConfig.getServletContext()).andReturn(servletContext).
anyTimes();

expect(request.getParameter("q")).andReturn(queryWord);

PrintWriter pw=new PrintWriter(System.out,true);
expect(response.getWriter()).andReturn(pw).anyTimes();
response.setContentType("text/html; charset=utf-8");
```

```
//重放 mock 对象
replay(request);
replay(response);
replay(servletConfig);
replay(servletContext);

instance.doPost(request, response);

pw.flush();

//检查预期和实际结果
verify(request);
verify(response);
verify(servletConfig);
verify(servletContext);
```

返回一个 JSON 数组格式的数据，其中汉字已经编码：

```
[{"c":1,"w":"PID"}, {"c":23,"w":"PLC"}, {"c":6,"w":"PLC \u897f\u95e8\u5b50"}, {"c":6,"w":"PLC\u57f9\u8bad"}, {"c":1,"w":"PLC\u5728\u7535\u529b\u7cfb\u7edf\u4e2d\u7684\u5e94\u7528"}, {"c":9,"w":"PLC\u7a0b\u5e8f"}, {"c":1,"w":"PLC\u7684\u9009\u578b"}, {"c":1,"w":"PLC5"}, {"c":2,"w":"PLC\u7f16\u7a0b\u624b\u518c"}, {"c":11,"w":"PLC \u63a7\u5236\u5668"}]
```

### 7.6.5 服务器端改进

当自动完成的性能需要进一步提高的时候，可以直接在内存中管理查询，而不是访问数据库取数。我们先设计词典格式，它由三列组成：第一列是词；第二列是搜索返回结果数量；第三列是用户搜索次数，中间用%隔开，例如：

综合教程第一册%34%2

搜索词是“综合教程第一册”，搜索返回结果数量是 34，用户搜索了 2 次。这样把用户搜索次数多的关键词放在前面优先显示。我们构造一个在“词典查找算法”部分已经介绍过的 Trie 树词典来实现快速的前缀匹配查找：

```
/**
 * 返回以一个前缀开始的所有关键词的数组
 *
 * @param prefix 前缀
 * @param numReturnValues 返回数组的最大长度
 * @return 返回数组结果
 */
```



```
public TSTItem[] matchPrefix(String prefix, int numReturnValues) {
    TSTNode startNode = getNode(prefix);
    if (startNode == null) {
        return null;
    }
    ArrayList<TSTItem> sortKeysResult = new ArrayList<TSTItem>();

    ArrayList<TSTItem> wordTable = sortKeysRecursion(
        startNode.EQKID,
        ((numReturnValues < 0) ? -1 : numReturnValues),
        sortKeysResult);
    int retNum = Math.min(numReturnValues, wordTable.size());

    Select.selectRandom(wordTable, wordTable.size(), retNum, 0);
    TSTItem[] fullResults = new TSTItem[retNum];
    for(int i=0; i<retNum; ++i) {
        fullResults[i] = wordTable.get(i);
    }

    return fullResults;
}

/**
 * 按顺序返回关键词，包括当前节点和与当前节点相关的所有节点对应的关键词。
 * 关键词将按顺序追加到结果的尾数
 * @param currentNode    当前节点
 * @param sortKeysNumReturnValues    最多返回结果数
 * @param sortKeysResult2    到目前为止的结果
 * @return    一个列表
 */
private ArrayList<TSTItem> sortKeysRecursion(
    TSTNode currentNode,
    int sortKeysNumReturnValues,
    ArrayList<TSTItem> sortKeysResult2) {

    if (currentNode == null) {
        return sortKeysResult2;
    }

    ArrayList<TSTItem> sortKeysResult =
        sortKeysRecursion(
            currentNode.LOKID,
            sortKeysNumReturnValues,
            sortKeysResult2);
```

```

if (currentNode.data != 0) {
    sortKeysResult.add(
        new TSTItem(getKey(currentNode),
            currentNode.data,
            currentNode.weight)
    );
}

sortKeysResult =
    sortKeysRecursion(
        currentNode.EQKID,
        sortKeysNumReturnValues,
        sortKeysResult);

return sortKeysRecursion(
    currentNode.HIKID,
    sortKeysNumReturnValues,
    sortKeysResult);
}

```

可以写一个简单的测试代码：

```

public static void main(String[] args) {
    SuggestDic sugDic = SuggestDic.getInstance();
    String prefix = "m";
    TSTItem[] ret = sugDic.matchPrefix(prefix, 10);
    for(TSTItem i:ret ) {
        System.out.println(i.key+":"+i.data+":"+i.weight);
    }
}

```

这样自动完成的 **Servlet** 类可以改写成下面这样：

```

String val = request.getParameter("query");

StringBuilder message = new StringBuilder("<ul>");
try {
    SuggestDic sugDic = SuggestDic.getInstance();

    TSTItem[] ret = sugDic.matchPrefix(val, 10);
    for(TSTItem i:ret ) {
        String Word = i.key;
        String Count = String.valueOf(i.data);

        message.append("<li style=\"font-size:12px;padding:0px;height:

```



```

15px;line-height:15px;overflow:hidden"><div style="text-align:left;float:left">");
message.append(Word);
message.append("</div><div style="text-align:right;float:right"><span class="informal">");
message.append(Count);
message.append("&nbsp;<font color=\"#009900">结果</font></span></div></li>");
}
} catch (Exception e) {
e.printStackTrace();
}

message.append("</ul>");

response.setContentType("text/html; charset=utf-8");
response.setCharacterEncoding("utf-8");
PrintWriter out = response.getWriter();

out.println(message.toString());

```

## 7.6.6 拼音提示

为了支持汉语拼音感应，需要把所有的词生成拼音列。Trie 树可以看成关键词和值的映射。拼音列和词本身都可以作为关键词，值这一列则存放词原型。例如对于“厦门”这个词，会存储两个关键词和值的映射。

xiamen -> 厦门

厦门 -> 厦门

这样当用户输入“厦”或“xia”都可能提示出“厦门”这个词。

对于基本的中文词提示来说，关键词和值都是一样。另外，注音程序把中文词转换成拼音，这部分数据支持汉语拼音感应功能。

因为存在多音字，按词注音会有好的结果。可以在 Trie 树的值域中存储一个词对应的拼音。

```

public static String yin(String sentence) { //传入一个字符串作为要处理的对象
int senLen = sentence.length(); //首先计算出传入的这句话的字符长度
int i = 0; //用来控制匹配的起始位置的变量

```

```

StringBuilder result = new StringBuilder(senLen);
TernarySearchTrie.MatchRet matchRet = new TernarySearchTrie.
MatchRet("", 0);
while (i < senLen){// 如果 i 小于此句话的长度就进入循环
    boolean match = dic.matchLong(sentence, i, matchRet);
    //正向最大长度匹配
    if (match){//已经匹配上, 按词注音
        i = matchRet.end;
        result.append(matchRet.data);
    } else//如果没有找到匹配上的词, 就按单字注音
    {
        result.append(ziYin.zi2Yin(sentence.charAt(i)));
        ++i;// 下次匹配点在这个字符之后
    }
}
return result.toString();
}

```

### 7.6.7 部署总结

提示词词典 suggestDic.txt 可以放在 WEB-INF/classes/dic/路径下。AutoComplete Servlet 可以放在 WEB-INF/lib/路径下, 通过 web.xml 发布。界面用到的 JavaScript 脚本 jquery.js、jquery.ajaxQueue.js、jquery.autocomplete.css 和 jquery.autocomplete.js 可以放在 ROOT/js 路径下。



## 7.7 集成其他功能

搜索引擎用户界面的一些功能还有: 对用户输入的拼写纠错提示、搜索结果的分类统计、根据用户搜索词返回相关搜索词、在搜索结果中再次查找、记录和统计搜索日志。

### 7.7.1 拼写检查

因为用户的查询本身是一个符合查询语法的字符串, 所以不能把用户的查询本身直接输入给拼写检查模块, 而要通过一个 didYouMeanParser 给出这个提示。拼写提示词的代码如下:

```

IndexSearcher is = new IndexSearcher(originalIndexDirectory);
Query query = didYouMeanParser.parse(queryString);
Hits hits = is.search(query);
String suggestedQueryString = null;

```





```
//如果搜索返回结果的数量小于阈值或者匹配第一个结果的分值小于最小值就查找提示词
if (hits.length() < minimumHits || hits.score(0) < minimumScore) {
    Query didYouMean = didYouMeanParser.suggest(queryString);
    if (didYouMean != null) {
        suggestedQueryString = didYouMean.toString(defaultField);
    }
}
is.close();
```

## 7.7.2 分类统计

首先定义分类统计信息类：

```
public class CatInf implements Comparable<CatInf> {
    public String no;//分类号
    public String name;//分类名
    public int count; //类别数量

    public int compareTo(CatInf obj){
        return (int)(obj.count - this.count);
    }
}
```

然后在页面执行搜索时执行如下过程：

```
/**
 * @param cat 当前类别编号
 * @param q 当前查询
 * @return 分类统计列表
 * @throws Exception
 */
public List<CatInf> catCounter(int cat, Query q) throws Exception {
    CategoryNode thisNode = ListContainer.catMap.get(String.valueOf(cat));
    if(thisNode.isLeaf) {
        //已经到达最后一级，不能再展开统计
        return null;
    }

    List<CategoryNode> children = thisNode.children;
    if(children == null) {
        return null;
    }
    ArrayList<CatInf> catList = new ArrayList<CatInf>( children.size() );
```

```

DocSetHitCollector all = new DocSetHitCollector(reader.maxDoc());
searcher.search(q, all );

DocSet allDocSet = all.getDocSet();

String termField = null;//层次列
if (cat<=0) {
    termField = "hs1";//第一层的 ID 号存储在 hs1 列
} else if (cat<100) {
    termField = "hs2";//第二层的 ID 号存储在 hs2 列
}
//如果还有后续层, 则按前缀匹配来搜
for (int i=0;i<children.size();++i)    {
    //统计每个子类的搜索结果数
    CategoryNode currentNode = children.get(i);
    DocSetHitCollector these = new DocSetHitCollector(reader.
maxDoc());

    searcher.search(
        new TermQuery(new Term(termField, currentNode.no)),
        these );
    int count = these.getDocSet().intersectionSize(allDocSet);
    if(count>0) {
        CatInf catInf = new CatInf();
        catInf.name = currentNode.name;
        catInf.no = currentNode.no;
        catInf.count = count;
        catList.add(catInf);
    }
}
Collections.sort(catList);//对分类统计结果排序后输出
return catList;
}

```

在 TagLib 中输出 table 标签中的分类统计结果:

```

public String getCatView() {
    if(_catList == null)
        return "";
    StringBuffer output = new StringBuffer();
    output.append("<table width=\"100%\" border=\"0\" cellspacing=\"0\"
cellpadding=\"2\">");
    output.append("<tr> ");
    int count =0 ;
    for(CatInf e : _catList) {

```



```

        output.append("<td><a href=\""); //URL 地址
        output.append(_url);
        output.append("?query=");
        output.append(_query); //查询词
        output.append("&");
        output.append(InitTag.CAT_KEY); //在 URL 中增加一个分类参数用于进一步导航
        output.append("=");
        output.append(e.no); //类别编号
        output.append("\" class=\"m\">");
        output.append(e.name); //类别名
        output.append("</a>");
        output.append(e.count); //该类别下的文档数量
        output.append("</td>");
        if(count%3 ==2) {
            output.append("</tr><tr>");
        }
        count++;
    }
    output.append(" </tr></table>");

    return output.toString();
}

```

最后 JSP 界面通过 Tag 调用 `getCatView`:

```
<list:prop property="catView"/>
```

### 7.7.3 相关搜索

一种方法是从搜索日志中挖掘字面相似的词作为相关搜索词列表。首先从一个给定的词语挖掘多个相关搜索词，可以用编辑距离为主的方法查找一个词的字面相似词，如果候选的相关搜索词很多，就要筛选出最相关的 10 个词。下面是利用 Lucene 筛选最相关词的方法。

```

private static final String TEXT_FIELD = "text";

/**
 *
 * @param words 候选相关词列表
 * @param word 要找相关搜索词的种子词
 * @return
 * @throws IOException
 * @throws ParseException
 */
static String[] filterRelated(HashSet<String> words, String word) {
    StringBuilder sb = new StringBuilder();
}

```

```

for(int i=0;i<word.length();++i){
    sb.append(word.charAt(i));
    sb.append(" ");
}

RAMDirectory store = new RAMDirectory();
IndexWriter writer = new IndexWriter(store, new StandardAnalyzer(),
true);

for(String text:words) {
    Document document = new Document();
    Field textField =
        new Field(TEXT_FIELD, text, Field.Store.YES, Field.Index.
            TOKENIZED);
    document.add(textField);
    writer.addDocument(document);
}
writer.close();

IndexSearcher searcher = new IndexSearcher(store);

QueryParser queryParser = new QueryParser(TEXT_FIELD,
                                           new StandardAnalyzer());
Query query = queryParser.parse(sb.toString());

Hits hits = searcher.search(query);
int maxRet = Math.min(10, hits.length());

String[] relatedWords = new String[maxRet];
for (int i = 0; i < maxRet ; i++) {
    Document document = hits.doc(i);
    String text = document.get(TEXT_FIELD);
    System.out.println(text);
    relatedWords[i]=text;
}
searcher.close();
store.close();

return relatedWords;
}

```

整理出这样的相关词表，第一列是关键词，后续是 10 个以内的相关搜索词：

集福轩婚礼%集福轩



手机定位跟踪系统%手机定位系统%手机定位%手机定位仪器  
喷绘材料卖店电话%我要喷绘材料卖店电话  
厦门房产%厦门租房%厦门新闻%厦门桑拿%房产%青岛房产%厦门%恒雄房产  
送水果%送水%水果  
三星传真机%三星手机

另外一种方法，可以把多个用户共同查询的词看成相关搜索词，需要有记录用户 IP 的搜索日志才能实现。然后通过 RelatedEngine 类查找某个关键词的相关词。

```
public static void main(String[] args) throws Exception {
    RelatedEngine re =new RelatedEngine(new File("D:/dic/
    relatedwords.txt"));
    String word = "徐家汇";
    String[] relatedWords = re.getRelated(word);
    for(String w : relatedWords) {
        System.out.println(w);
    }
}
```

输出相关搜索词如下所示：

上海徐家汇  
徐汇  
徐家汇价格是  
上房徐家汇路附近是吗

最后通过自定义的 Tag 标签 RelatedTag 在 JSP 页面显示出相关搜索词。

在标签库描述符中定义 Tag：

```
<tag>
  <name>relatedWords</name>
  <tag-class>com.bitmechanic.listlib.RelatedTag</tag-class>
  <description></description>

  <attribute>
    <name>index</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>

  <attribute>
```

```

        <name>url</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

    <attribute>
        <name>query</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

</tag>

```

最后在 JSP 页面中引用标签：

```

<list:relatedWords index="D:/search/related" url="Search.jsp" query="
<%=query%>"/>

```

#### 7.7.4 再次查找

经常需要从结果中缩小范围再次查找信息。一个实现方法是通过“+”连接符连接上次查询和当前查询。例如：`inputstr` 记录了上次查询词，`queryString` 记录当前查询词，实现代码如下所示：

```

if (refind) //如果需要再次查找
    queryString = " + (" + queryString + ") + (" + inputstr + ")";

```

使用这个新的查询词就可以实现再次搜索的功能。

#### 7.7.5 搜索日志

搜索日志是用来分析用户搜索行为和信息需求的重要依据。一般记录如下信息：

- 搜索关键字；
- 用户来源 IP；
- 本次搜索返回结果数量；
- 搜索时间；
- 其他需要记录的应用相关信息。

IP 地址是最容易获取的信息，但其局限性也较为明显：伪 IP、代理、动态 IP、局域网共享同一公网 IP 出口……这些情况都会影响基于 IP 来识别用户的准确性，所以 IP 识别用



户的准确性比较低，目前一般不会直接采用 IP 来识别用户。

可以通过 Cookie 记录用户 ID。Cookie 是从用户端存放的 Cookie 文件记录中获取的，这个文件里面一般在包含一个 Cookieid 的同时也会记下用户在该网站的 userid（如果你的网站需要注册登录并且该用户曾经登录过你的网站且 Cookie 未被删除），所以在记录日志文件中 Cookie 项的时候可以优先查询 Cookie 中是否含有用户 ID 类的信息，如果存在则将用户 ID 写到日志的 Cookie 项，如果不存在则查找是否有 Cookieid，如果有则记录，没有则记为“-”，这样日志中的 Cookie 就可以直接作为最有效的用户唯一标识符被用作统计。当然这里需要注意该方法只有网站本身才能够实现，因为用户 ID 作为用户隐私信息只有该网站才知道其在 Cookie 的设置及存放位置，第三方统计工具一般很难获取。

通过以上的方法实现用户身份的唯一标识后，我们可以通过一些途径来采集用户的基础信息、特征信息及行为信息，然后为每位用户建立起详细的 Profile，具体途径有：

- 用户注册时填写的用户注册信息及基本资料；
- 从网站日志中得到的用户浏览行为数据；
- 从数据库中获取的用户网站业务应用数据；
- 基于用户历史数据的推导和预测；
- 通过直接联系用户或者用户调研的途径获得的用户数据；
- 由第三方服务机构提供的用户数据。

通过用户身份识别及用户基本信息的采集，我们可以通过网站分析的各种方法在网站实现一些有价值的应用：

- 基于用户特征信息的用户细分；
- 基于用户的个性化页面设置；
- 基于用户行为数据的关联推荐；
- 基于用户兴趣的定向营销。

为了不影响即时搜索的速度，一般不把搜索日志记录直接记录在数据库中，而是写在文本文件中，可以使用 logback（<http://logback.qos.ch/>）的日志功能实现。先加 slf4j 的包 slf4j-api-1.6.1.jar，然后加 logback-classic 和 logback-core 包，配置 logback.xml。

这里把当前日志写到 D:/logs/log 文件中，新一天的日志开始的时候，昨天的日志生成一个新文件。

在搜索类中初始化日志类：

```
private static Logger logger = Logger.getLogger(SearchBbs.class.getName());
```

然后当用户执行一次搜索时，记录：

```
logger.info(_query+"|"+desc.count+"|"+bbs+"|"+ip);
```

日志文件 log.txt 记录的结果例子如下：

什么是新生儿|37|topic|124.1.0.0|2007-11-21 12:25:36

什么是新生儿|28|bbs|124.1.0.0|2007-11-21 12:25:42

怀孕|18|topic|124.1.0.0|2007-11-21 12:26:05

怀孕|2|shangjia|124.1.0.0|2007-11-21 12:26:05

怀孕|145|bbs|124.1.0.0|2007-11-21 12:26:06

怀孕|18|topic|124.1.0.0|2007-11-21 12:30:33

这里，第一列是用户搜索词；第二列是搜索返回结果数量；第三列是搜索类别；第四列是 IP 地址；第五列是搜索的时间。

然后定义搜索日志统计表，例如我们需要统计搜索最多的词，可以把搜索最多的词放在 keywordAnalysis 表中：

```
CREATE TABLE [keywordAnalysis] (
    [searchTerms] [varchar] (50) NOT NULL ,--搜索词
    [AccessCount] [int] NULL , --搜索计数
    [Result] [int] NULL --该词返回结果数
)
```



## 7.8 搜索日志分析

因为搜索关键词是用户输入的文本信息，所以可以从搜索日志中了解用户使用搜索的意图。有人把 Google 的搜索关键词排行榜称作人类意图数据库，这来源于对搜索日志的分析。

### 7.8.1 日志信息过滤

公开的搜索会有很多爬虫的访问。搜索日志中包括大量的 Google 爬虫信息，需要把它和普通用户的搜索区分出来。





可以从请求的信息中判断出是哪一种爬虫。

- Baidu 爬虫的 “User-Agent” 信息：

```
Baiduspider+(+http://help.baidu.jp/system/05.html)
```

- Goolge 爬虫的 “User-Agent” 信息：

```
Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)
```

下面是具体程序实现：

```
String userAgent = request.getHeader( "User-Agent" );

public static String[] getBotName(String userAgent) {
    userAgent = userAgent.toLowerCase();
    int pos=0;
    String res=null;
    if ((pos=userAgent.indexOf("google/"))>-1) {
        res= "Google";
        pos+=7;
    } else
    if ((pos=userAgent.indexOf("msnbot/"))>-1) {
        res= "MSNBot";
        pos+=7;
    } else
    if ((pos=userAgent.indexOf("googlebot/"))>-1) {
        res= "Google";
        pos+=10;
    } else
    if ((pos=userAgent.indexOf("webcrawler/"))>-1) {
        res= "WebCrawler";
        pos+=11;
    } else
    if ((pos=userAgent.indexOf("inktomi"))>-1) {
        res= "Inktomi";
        pos=-1;
    } else
    if ((pos=userAgent.indexOf("teoma"))>-1) {
        res= "Teoma";
        pos=-1;
    }
    if (res==null) return null;
    return getArray(res,res,res + getVersionNumber(userAgent,pos));
}
```

7.8.2 信息统计

搜索热词的统计界面如图 7-9 所示。

导出文本	查询条件	2008-07-07	至	2008-07-07	统计
关键词	搜索次数	返回结果数			
下半年运势	4905	7			
内裤	1610	5100			
内衣	1367	10529			
裙子	1176	12171			
测试	1060	20			
凉鞋	929	2569			
搬上床	854	2			
恤	556	9386			
雪纺	525	3721			
12星座	496	1326			
凉鞋	456	503			
男人最想娶的十类女人	440	1			
吊带	438	20			
袜子	384	20			
发型	368	12414			
牛仔	309	12957			
女人	292	94979			
性爱	289	9339			
短袖	279	3151			
健美	273	10976			
小计：	17,006	本页返回结果平均数: 9,459			
总计：	51,964	总共返回结果平均数: 1,577			

图 7-9 搜索热词的统计页面

按地区来源的搜索日志分析界面如图 7-10 所示。

地区	搜索次数	总排名
广东省	97716	1
浙江省	49764	2
山东省	47406	3
河北省	43495	4
北京市	41774	5
江苏省	40858	6
辽宁省	36944	7
湖北省	33046	8
河南省	31875	9
四川省	31498	10
湖南省	31243	11
陕西省	27327	12
安徽省	26522	13
上海市	25835	14
山西省	24903	15
广西省	24664	16
黑龙江省	24001	17
福建省	22834	18
江西省	21617	19
吉林省	18048	20

图 7-10 按地区来源的搜索日志分析页面

按地区统计搜索词需从用户的访问 IP 查询出对应的地址。如表 7-3 所示的 IP 地址表记录了一个地区的 IP 范围。

表 7-3 IP 地址表

ip1	ip2	country	city	countryNo	provinceNo
3740518268	3740518268	湖南省永州市	金鹰网吧	1	12
3740518269	3740518400	湖南省永州市	电信	1	12
3740518401	3740518401	湖南省永州市祁阳县	怡心苑网吧	1	12
3740518402	3740518417	湖南省永州市	电信	1	12

下面的实例代码返回用户 IP 所在省：

```
String ipin[] = getIp.split("\\.");
long ipinfo[] = new long[4];
for (int i = 0; i < ipinfo.length; i++) {
    //从 String 类型的 IP 地址到 long 型的转换可以用查表法更快的实现
    ipinfo[i] = Integer.parseInt(ipin[i]);
}
long num=ipinfo[0] * 256 * 256 * 256 + ipinfo[1] * 256 * 256 + ipinfo[2]
* 256 + ipinfo[3] - 1;
String sql = "select DIC_Province.caption from ip,DIC_Province where
ip.provinceNo = DIC_Province.id and ip.ip1<=" + num + " and
ip.ip2>= " + num + " ";
st = con.createStatement();
rs = st.executeQuery(sql);
if (rs.next()) {
    area = rs.getString(1);
}
```

首先建立搜索日志统计表：

```
CREATE TABLE SC_SEARCH_STAT (
    ID NUMERIC(12 , 0) IDENTITY ,           //自增长 ID
    SEARCH_WORD VARCHAR(90) NULL,           //搜索词
    SEARCH_NUM INT NULL,                     //搜索次数
    SEARCH_DATE DATE NULL,                  //搜索日期
    SEARCH_RESULT INT NULL                   //搜索词的返回结果数量
)
```

搜索统计表每天更新一次。每次把上一天的搜索日志文件中的数据统计后写入该表。搜索次数 SEARCH\_NUM 指一天内搜索 SEARCH\_WORD 这个词的独立 IP 的统计数量，同日同地址同搜索词只算一次。搜索统计用到的主要数据结构有：

```
HashMap<String,HashSet<String>> word2IP =
    new HashMap<String,HashSet<String>>(); //搜索词到 IP 的映射
HashMap<String,Integer> word2ResultNum =
```

```

        new HashMap<String, Integer>(); //搜索词到搜索结果数的映射
...
//统计信息
HashSet<String> ips = word2IP.get(key); //如果搜索词已存在
if (ips!=null) {
    ips.add(strIP); //增加当前 IP
    word2ResultNum.put(key, resultNum);
} else if (resultNum > 0) { //如果搜索返回结果数大于零
    ips = new HashSet<String>();
    ips.add(strIP);
    word2IP.put(key, ips);
    word2ResultNum.put(key, resultNum);
}
...
//统计信息写入统计表
String sql =
"insert into SC_SEARCH_STAT (SEARCH_WORD, SEARCH_NUM, SEARCH_DATE, SEARCH_
RESULT) values(?, ?, ?, ?)";
PreparedStatement pstmt = con.prepareStatement(sql);

for (Entry<String, HashSet<String>> e : word2IP.entrySet()) {
    pstmt.setString(1, e.getKey());
    pstmt.setInt(2, e.getValue().size());
    pstmt.setString(3, yesDate);
    pstmt.setInt(4, word2ResultNum.get(e.getKey()));
    pstmt.executeUpdate();
}

```

### 7.8.3 挖掘日志信息

可以从不同的角度挖掘搜索日志。

- 挖掘单个词。可以挖掘出用户对查询语法的使用情况，例如 filetype、site 等查询语法。可以统计用户搜索中使用空格分开多个词来搜索的比例。
- 按用户会话（session）挖掘词。有用户先搜索“工程电磁学基础（第6版）”，没有结果返回，几秒钟后他换了一个搜索词“工程电磁学基础”，这次有22条结果返回。因此可以把用户对查询词的修改分为4种情况：减少查询词、增加查询词、部分替换查询词、完全更换查询词。另外，还可以统计用户搜索词之间的词序，考虑上一个词到下一个词之间的转移概率。
- 根据用户与词之间的关联可以挖掘出词与词之间的关联或者用户与用户之间的关联。观察到一个用户搜索“眉笔”后，又搜索了“粉底液”和“紧肤水”，这些词都是与美容相关的商品名，所以考虑使用关联规则挖掘出相关词。可以根据用户的



搜索词对用户分类，计算出用户对每个类别的隶属度。



## 7.9 本章小结

本章介绍了使用 JSP 实现的搜索界面。并且介绍了很多重要而且基本的搜索功能界面实现，例如复杂条件搜索界面和用户输入提示词、分类查找界面等。

如果把搜索功能作为单独的一个域名，例如 <http://so.lietu.com>，不要忘记加网站头像：favicon.ico 文件。



## 第 8 章

# 使用 Solr 实现企业搜索

Lucene 仅仅是一个全文检索包，不是一个独立的搜索服务。如果在异构环境直接调用 Lucene 可能会导致系统不稳定，例如开发 Web 界面的工具采用 PHP 时，采用桥接方式调用 Lucene 容易导致内存泄露。本章介绍的 Solr 搜索服务器就是一个支持多种前台开发语言的搜索框架。Solr 提供了 XML 接口的全文检索服务，支持 ASP.NET、JSP、PHP、Python 等多种前台开发语言。



### 8.1 Solr 简介

Solr 把对 Lucene 索引的调用和管理做了一个 REST 风格的封装。它是一个 Web 方式的索引服务器。需要搜索的应用可以通过 http 请求调用 Solr 索引服务。可以把 Solr 看成类似 MySQL 的数据库系统，在配置文件 schema.xml 中定义表结构。和用 JDBC 访问数据库不同，通过 HTTP 的 POST 方式发送需要增加的数据给 Solr，通过 HTTP 的 GET 方式查询 Solr 管理的索引库。外部程序首先把 HTTP 请求提交给类似 Tomcat 或者 Resin 的 Web 服务器，然后 Solr 内部包含 Servlet 来写数据到索引库或者查询索引库。Solr 的内部结构如图 8-1 所示。

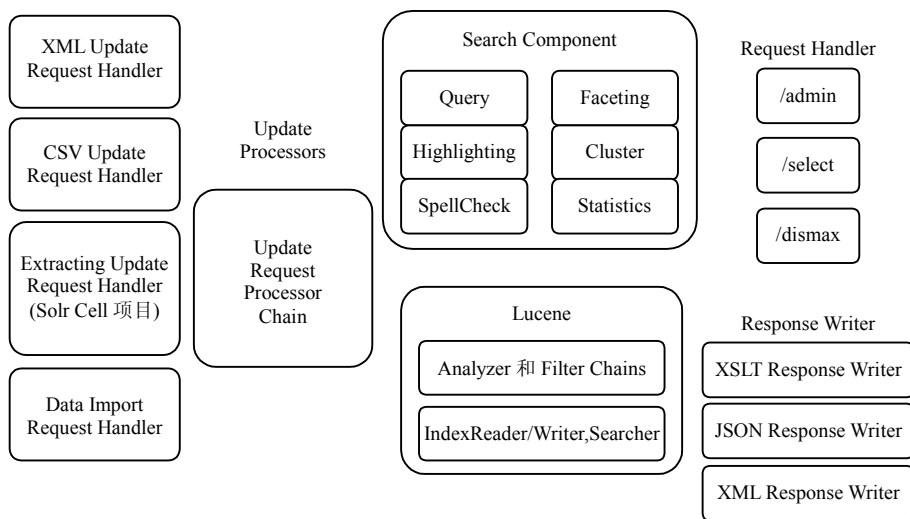


图 8-1 Solr 内部结构图

Solr 是网站内部系统演化出来的一个基于 Lucene 的开源项目。在 2004 年秋天，CNET 启动 Solr 项目的前身 Solar。2005 年夏天，CNET 产品目录搜索开始使用 Solar。2006 年 1 月 CNET 把这个搜索项目代码捐赠给 Apache 并命名为 Solr。2007 年 1 月 Solr 毕业成为 Lucene 的子项目并发布 1.2 版本。2008 年 9 月发布 1.3.0 版本。2009 年 11 月发布 1.4 版本。Solr 当前主要由 Yonik 维护代码，Yonik 是斯坦福大学计算机专业硕士毕业，关于 Solr 的文档介绍在 <http://wiki.apache.org/solr/>。

单台机器的计算能力有限，可以采用 Solr 搭建多机集群的分布式搜索来实现高负载和高可用性。一个完整的分布结构如图 8-2 所示。

本书中介绍的 Solr 是 1.4 版本，从 Solr 的官方网站 <http://lucene.apache.org/solr/> 可以下载到这个版本和当前最新的版本。



## 8.2 Solr 基本用法

Solr 自身由服务器端和客户端组成。服务器端提供搜索服务，客户端提供 Web 界面开发支持。

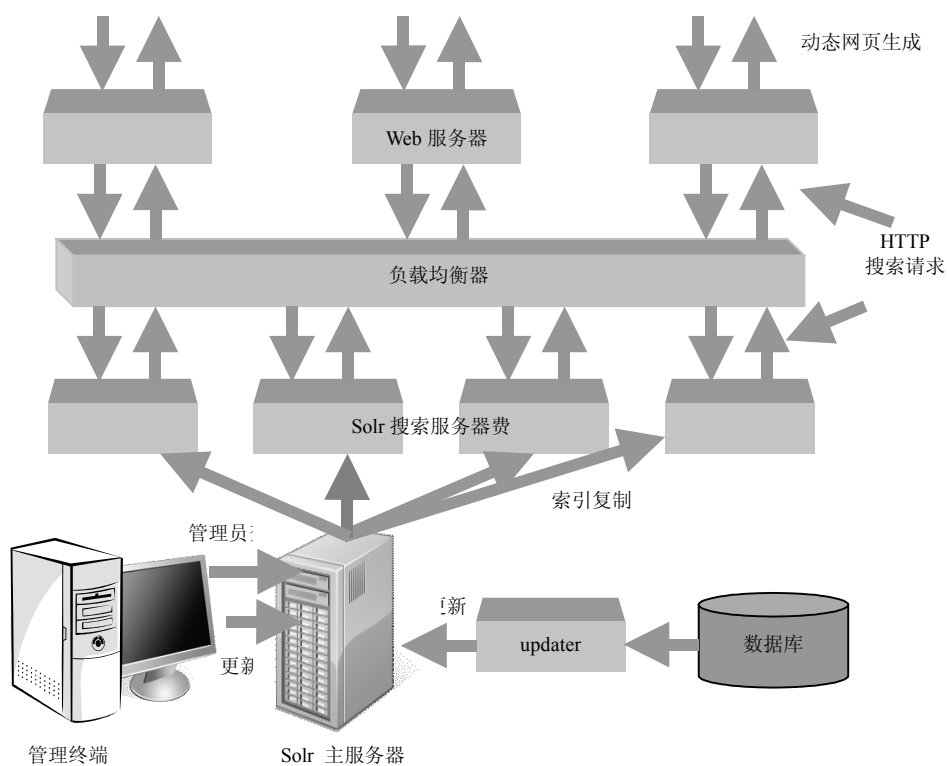


图 8-2 Solr 的完整分布结构图

### 8.2.1 Solr 服务器端的配置与中文支持

可以把 Solr 的服务器端看成是包装在 Web 应用服务器中的 Lucene 服务器。和 Lucene 索引库相比的好处是可以实现缓存预加载。Web 应用服务器通常可以选用 Resin 或者 Tomcat。

首先保证 Web 服务器的内存使用量。以 Resin 的 3.1 版本为例，修改 resin.conf 中关于 JVM 的配置，增加最大内存使用量到 500MB：

```
<jvm-arg>-Xmx500m</jvm-arg>
```

服务器主要通过 JNDI 中的 solr/home 来指定 Solr 存储库的路径。例如，在 Resin 中 resin.conf 的内容如下：

```
<web-app id="/index" document-directory="webapps/index">
  <env-entry>
    <env-entry-name>solr/home</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
```

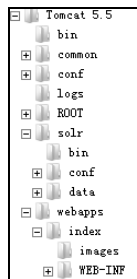


```

        <env-entry-value>${resin.home}/solr</env-entry-value>
    </env-entry>
</web-app>

```

从分布路径上来说，Solr 由两部分组成。一部分是后台路径，可以放在 Web 应用服务器根路径的 solr 子路径下，其中的 conf 子路径存放配置文件，其中的 data 子路径存放索引数据；还有一部分是 Web 管理界面，存放在 webapps 目录下。



Tomcat 的最简单配置如图 8-3 所示。

直接把 Solr 配置路径拷贝到 Tomcat 的子目录下就可以。初次配置 Solr 的时候，建议在命令行运行 Tomcat，这样容易发现错误。

另外也可以通过 Java 虚拟机的系统属性 solr.solr.home 来指定存储库路径。

```

#export JAVA_OPTS="$JAVA_OPTS -Dsolr.solr.home=/my/custom/solr/home/
dir/"

```

有时候需要建立好几个不同格式的索引。例如垂直行业搜索中需要分别搜索产品和公司，需要把产品信息和公司信息放在不同的索引格式中。同一个 Tomcat 下可以配置多个 Solr 实例来对应不同的索引格式。例如要配置两个 Solr 实例：company 和 product，可以在 /usr/local/tomcat55/conf/Catalina/localhost 路径下建立两个配置文件：company.xml 和 product.xml，分别对应两个 Web 应用，其中 product.xml 的内容如下所示：

```

#cat ./product.xml
<Context docBase="" debug="0" crossContext="true" >
    <Environment name="solr/home" type="java.lang.String"
        value="/usr/local/product" override="true" />
</Context>

```

和部署在同一个 Web 服务器中的多个 Web 应用不同，Solr 中的 multi core 功能指在同一个 Web 应用中管理多个索引。在并发量很少的情况下，multi core 在性能上优势较为明显，但当并发量增加后，multi core 响应速度反而比标准配置慢。因为 multi core 用得较少，所以这里不介绍。

为了支持中文，Tomcat 中的相关配置如下所示，在 server.xml 中增加对 UTF-8 的处理，这样是为了支持 HTTP 的 GET 方法。

```
<Connector port="80" useBodyEncodingForURI="true" URIEncoding="UTF-8" />
```

useBodyEncodingForURI="true"的意思是用页面的编码去处理 POST 请求。另外在 Solr 的 Web 项目的 web.xml 配置中增加对 UTF-8 的转码:

```
<filter>
  <filter-name>Set Character Encoding</filter-name>
  <filter-class>filters.SetCharacterEncodingFilter
</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>utf-8</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Set Character Encoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

增加这个 Filter 是为了支持 HTTP 的 POST 方法。SetCharacterEncodingFilter 类可以在 webapps\examples\WEB-INF\classes 路径下找到。

Solr 中有以下两个在 conf 路径下的配置文件。

- solrconfig.xml: 用来配置 Solr 运行的系统参数, 例如, 缓存, 插件等。
- schema.xml: 主要定义索引结构相关的信息, 例如, types、fields 和其他的一些默认设置。

然后需要调整的是 solrconfig.xml。

```
<maxWarmingSearchers>4</maxWarmingSearchers>
```

自动加载缓存的线程数默认是 4, 可以调整小一点, 最好少于 CPU 核数量, 则可能使一个双 CPU 双核的机器 CPU 使用率达到近 100%。

schema.xml 配置文件定义了 Solr 中索引库的结构, 可以把它类比作数据库中的一张表。可以通过 field 来定义列, fieldType 定义列类型, uniqueKey 指定索引库的唯一标识列 (类似数据库表中的主键), defaultSearchField 指定默认搜索列。schema.xml 使用的基本的数据类型如下所示:



```
<fieldtype name="sint" class="solr.SortableIntField" /> //整型
<fieldtype name="slong" class="solr.SortableLongField"/> //长整型
<fieldtype name="sfloat" class="solr.SortableFloatField"/> //浮点数
<fieldtype name="sdouble" class="solr.SortableDoubleField" /> //双精度
<fieldtype name="date" class="solr.DateField" /> //日期
```

和 Lucene 比较起来，这些数据类型有更优化的存储方式。

下面定义一个基本的文字搜索列类型。

```
<fieldtype name="text_ws" class="solr.TextField" positionIncrementGap=
"100">
<analyzer>
    <tokenizer class="CnTokenizerFactory"/> CnTokenizerFactory
</analyzer>
```

这里的 CnTokenizerFactory 是一个分词类，它扩展了 BaseTokenizerFactory。当然也可以定义自己的做 Tokenizer 的类。所有这样的类都必须是 BaseTokenizerFactory 的子类。

```
package org.apache.solr.analysis;
public class CnTokenizerFactory extends BaseTokenizerFactory{
    public TokenStream create(Reader input) {
        return new CnTokenizer(input);
    }
}
```

在 schema.xml 中定义了索引库的结构，下面定义三列，分别是唯一 id、标题和内容。

```
<field name="id" type="string" indexed="true" stored="true" multi
Valued="false" />
<field name="title" type="text_ws" indexed="true" stored="true" multi
Valued="false" />
<field name="body" type="text_ws" indexed="true" stored="true" multi
Valued="false" />
```

在这里，multiValued="false"的意思是一列只允许存储一个值。如果需要一列存储多个值，例如一个商品属于多个类别（cat），则定义：

```
<field name="cat" type="string" indexed="true" stored="true" multi
Valued="true" />
```

定义 id 为唯一标识列（id 即为上面定义的名字是 id 的列）。

```
<uniqueKey>id</uniqueKey>
```

定义 body 为默认搜索列。

```
<defaultSearchField>body</defaultSearchField>
```

它的服务器端管理界面包含 Solr 和索引库中的一些宏观的参数。相比较而言, Luke 则显示更细一些的 Document 和 Term 等信息。<http://localhost/solr/admin/>是一个已经安装好的 Solr 界面。可以先通过 <http://localhost/solr/admin/analysis.jsp?highlight=on> 来对每个列做基本的测试,看看对定义好的 title 列是否能对中文做正确的处理。

有时候需要调试某个查询词是否能匹配上某个文档内容,你可能会困惑于一个查询词不能匹配上某个文档,就好像一把钥匙打不开一把锁,所以可以把查询词和文档看成是钥匙和锁的关系,查询词是钥匙,而待查找的文档则是锁。当查询词正好匹配上文档,感觉像用钥匙打开正确的锁一样。图 8-4 的调试界面有助于找出问题所在。

**Solr Admin (example)**  
localhost.localdomain:8080  
cwd=/usr/local/resin-3.1.0 SolrHome=solr/

**Field Analysis**

Field name: title

Field value (Index): 我爱北京天安门

verbose output ☒

highlight matches ☒

Field value (Query): 天安门

verbose output ☒

Analyze

**Index Analyzer**  
org.apache.solr.analysis.CnTokenizerFactory

term position	1	2	3	4	5
term text	我	爱	北	京	天
term type	r	v	ns	ns	ns
source start,end	0,1	1,2	2,4	4,6	6,7

**Query Analyzer**  
org.apache.solr.analysis.CnTokenizerFactory

term position	1	2
term text	天	安
term type	ns	ns
source start,end	0,2	2,3

图 8-4 Solr 的 Web 管理界面

顶部灰色部分介绍如下。

- 头部信息,当启动多个 Solr 实例时,可以帮助了解在操作哪个实例。IP 地址和端口号都是可见的。
- example (Admin 旁边)是对这个 schema 的引用,仅仅是标识这个 schema。如果你有很多 schema,可以用这个标识去区分。
- 当前工作目录 (cwd) 和 Solr 的根目录 (SolrHome)。



在 Tomcat 中配置多个 Solr 实例的方法是，用 JNDI 的方法配置多个 solr.home。例如下面在 \$CATALINA\_HOME/conf/Catalina/localhost 下为每个 Solr 实例的 webapp 创建独立的 context：

```
$ cat /tomcat55/conf/Catalina/localhost/solr1.xml
<Context docBase="" debug="0" crossContext="true" >
  <Environment name="solr/home" type="java.lang.String" value="f:
    /solr1home" override="true" />
</Context>
$ cat /tomcat55/conf/Catalina/localhost/solr2.xml
<Context docBase="" debug="0" crossContext="true" >
  <Environment name="solr/home" type="java.lang.String" value="f:
    /solr2home" override="true" />
</Context>
```

把每个 Solr 实例服务器端配置都放在不同的目录，修改 solrconfig.xml 的索引数据存储路径：

```
<dataDir>${solr.data.dir:./solr1/data}</dataDir>
```

solrconfig.xml 文件记录一个 Solr 实例的配置信息。如上述索引数据存储路径，还可以配置自动加载缓存的线程数量：

```
<maxWarmingSearchers>2</maxWarmingSearchers>
```

自动加载缓存的一个线程数默认是 4，有时候要调整小一点，否则会使一个双核的机器 CPU 使用率达到近 100%。

还可以在 solrconfig.xml 中定义默认搜索处理类及其他 Handler 信息等。

默认搜索处理器 standard Handler 的定义如下所示：

```
<requestHandler name="standard" class="solr.SearchHandler" default="true">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
  </lst>
</requestHandler>
```

有个小问题是 Solr 的日志在默认情况下增加的很快，可以通过日志的输出级别来控制日志输出，Solr 中的配置界面是 <http://localhost/solr/admin/logging.jsp>。

## 8.2.2 把数据放进 Solr

可以通过 `post.jar` 把数据放到 Solr 存储库。`post.jar` 读取的是 XML 文件。如果处理中文，这个 XML 文件必须是 UTF-8 编码的，否则就会出现乱码。下面是一个 XML 文件的样例。

```
<add>
  <doc>
    <field name="id">126788</field>
    <field name="title">中文内容测试,TEST</field>
    <field name="body">上海</field>
  </doc>
</add>
```

我们通过客户端程序 `SolrJ` 把数据库中的数据转换成符合 Solr 格式的 XML 数据流，然后通过 HTTP 发送出去。

```
SolrServer server = new HttpSolrServer("http://localhost:
8983/solr/rss");
Random rand = new Random();
//索引一些文档
Collection<SolrInputDocument> docs = new HashSet<SolrInputDocument>();
for (int i = 0; i < 10; i++) {
    SolrInputDocument doc = new SolrInputDocument();
    doc.addField("link", "http://non-existent-url.foo/" + i + ".html");
    doc.addField("source", "Blog #" + i);
    doc.addField("source-link", "http://non-existent-url.foo/index.html");
    doc.addField("subject", "Subject: " + i);
    doc.addField("title", "Title: " + i);
    doc.addField("content", "This is the " + i + "(th|nd|rd) piece of
content.");
    doc.addField("category", CATEGORIES[rand.nextInt(CATEGORIES.
length)]);
    doc.addField("rating", i);
    System.out.println("Doc[" + i + "] is " + doc);
    docs.add(doc);
}
UpdateResponse response = server.add(docs);
System.out.println("Response: " + response);
```

数据更新以后，如果要在索引库即刻看到更新的数据，需要提交更新：

```
<Commit/>
```

在 `SolrJ` 中调用执行 `server.commit()` 来更新数据。



除了显式调用 `commit`，同时 Solr 也支持自动提交数据。可以在 `solrconfig.xml` 配置文件中修改自动提交的条件：

```
<!-- 当满足以下条件时执行自动提交：
      maxDocs - 自从上次提交以来，更新数量超过指定次数时提交更新
      maxTime - 当超过指定时间(ms 为单位)后提交
-->
<autoCommit>
  <maxDocs>10000</maxDocs>
  <maxTime>100000</maxTime>
</autoCommit>
```

如果不需要自动提交数据，只是通过程序显式地提交，可以把相关的值都置成负数：

```
<autoCommit>
  <maxDocs>-1</maxDocs>
  <maxTime>-1</maxTime>
</autoCommit>
```

优化索引使用如下的 Solr 命令：

```
<Optimize/>
```

在 SolrJ 中调用执行 `server.optimize()` 来更新数据。

可以编写一个后台独立运行的程序来同步数据库和索引，把数据库中的数据增量放入索引库中。

```
public static void main(String[] args) throws Exception, InterruptedException {
    long sleepTime = 5000L; //5 秒
    for(;;){
        boolean findNew = indexDb(); //查看数据库是否有新数据要索引

        if(findNew) {
            sleepTime = 5000L;
        }
        else {
            if(sleepTime < 500000L) {
                sleepTime = sleepTime*2; //没有新数据则延长检测新数据的间隔
            }
        }
        System.out.println("sleep..." + sleepTime);
    }
}
```

```

        Thread.sleep(sleepTime);
    }
}

```

考虑 `indexDb()` 如何实现，也就是如何从待索引的表中找出要索引的新数据。可以根据表中的唯一 `id` 列查询出要索引的新数据。

### 8.2.3 删除数据

可以通过查询的方式删除 Solr 的索引数据。例如根据查询删除数据：

```

#curl http://192.168.10.30:8080/solr/update
--data-binary "<delete><query>id:314685</query></delete>" -H "Content
-type:text/xml"
#curl http://localhost/gongkong/update
--data-binary "<delete><query>id:http\:\\\\www.aljoin.com\\news\\
2007-10\\20071010154644.htm</query></delete>" -H "Content-type:text/xml"

```

或者直接通过 `Id` 列删除：

```

#curl http://192.168.10.30:8080/solr/update --data-binary "<delete>
<id>314685</id></delete>" -H "Content-type:text/xml"

#curl http://localhost/gongkong/update --data-binary "<delete><id>http
\\:\\\\www.aljoin.com\\news\\2007-10\\20071010154644.htm</id></delete>"
-H "Content-type:text/xml"

```

删除所有标题或者内容中含有“答案”这个关键字的结果：

```

#curl http://192.168.10.30:8080/solr/update --data-binary "<delete>
<query>title:答案 OR body:答案</query></delete>" -H "Content-type:text/xml"

```

SolrJ 删除数据的例子：

```

String url = "http://localhost/solr/";
SolrServer server = new HttpSolrServer( url );
String q = "source:\"industrial acs\"";
UpdateResponse res = server.deleteByQuery( q );
System.out.println(res.getStatus()); //取得返回状态值
server.commit( true, true );

```

如果要清空所有的数据，可以把 Solr 服务停止后直接删除 Solr 的索引路径，Solr 启动后会重建索引路径。





## 8.2.4 Solr 客户端与搜索界面

一般通过各种动态页面生成服务器返回搜索结果给查询用户。常用的动态网页开发工具有 ASP.NET、JSP 或 PHP 等。Solr 提供对应的客户端来支持各种常用开发工具。Solr 返回的数据格式通常是 XML，返回 JSON、Python、Ruby 格式的数据也是支持的。Solr 客户端是对 HTTP 请求的封装，同时也包括了对 XML 格式搜索结果的解析。已有的部分 Solr 客户端如表 8-1 所示。

表 8-1 Solr 客户端列表

前台语言	Solr 客户端	网 址
JSP	SolrJ	<a href="http://svn.apache.org/repos/asf/lucene/solr/trunk/src/solrj/">http://svn.apache.org/repos/asf/lucene/solr/trunk/src/solrj/</a>
PHP	solr-php-client	<a href="http://code.google.com/p/solr-php-client/">http://code.google.com/p/solr-php-client/</a>
ASP.NET	SolrNet	<a href="http://code.google.com/p/solrnet/">http://code.google.com/p/solrnet/</a>
AJAX	ajax-solr	<a href="http://evolvingweb.github.com/ajax-solr/">http://evolvingweb.github.com/ajax-solr/</a>
Ruby on Rails	acts_as_solr	<a href="http://acts-as-solr.rubyforge.org/">http://acts-as-solr.rubyforge.org/</a>

SolrJ 是一个易于使用的客户端。因为 SolrJ 除了支持 XML 格式，还可以采用 Java 二进制（wt=javabin）方式返回搜索结果，所以性能相对其他客户端更好。下面是 SolrJ 一个简单的使用例子。

```
String url = "http://hot.lietu.com:8080/solr/";
SolrServer server = new HttpSolrServer( url );

SolrQuery query = new SolrQuery("NBA");

QueryResponse response = server.query( query );
//取得符合查询条件的总数
int numFound = response.getResults().getNumFound();

for(SolrDocument d : response.getResults()){
    String id = (String)d.getFieldValue("id");
    System.out.println("id:"+id);
    String title = (String)d.getFieldValue("title");
    System.out.println("title:"+title);
}
```

因为 Solr 内部已经有分页控制，所以实际显示的记录会比搜索到的结果少。分页程序需要计算总共有多少页以及当前在第几页。下面增加对高亮显示的支持。

```
String url = "http://hot.lietu.com:8080/solr/";
SolrServer server = new HttpSolrServer( url );
```

```

SolrQuery query = new SolrQuery("NBA");

HighlightingParams hp = query.getHighlightingParams();
hp.addField("body");

hp.setSimplePre("<font color=red>");
hp.setSimplePost("</font>");
//限制高亮显示词的范围
hp.setHRequireFieldMatch(true);
QueryResponse response = server.query( query );
System.out.println("found:"+response.getResults().getNumFound());

for(SolrDocument d : response.getResults()){
    String id = (String)d.getFieldValue("id");
    System.out.println("id:"+id);
    //取得高亮显示的第一段
    String hl = response.getHighlighting().get(id).get( "body" ).
        get(0);
    System.out.println("hl:"+hl);
}

```

下面是对“catlid”这一列的分类统计：

```

String url = "http://hot.lietu.com:8080/solr/";
SolrServer server = new HttpSolrServer( url );

SolrQuery query = new SolrQuery("NBA");

query.getFacetParams().addField("catlid");

QueryResponse response = server.query( query );
System.out.println("found:"+response.getResults().getNumFound());

List<Count> facetCounts = response.getFacetField("catlid").getValues();
for(Count c:facetCounts ){//打印分类统计结果
    System.out.println(c);
}

```

VelocityResponseWriter 可以返回从 Velocity 模版生成的内容。

## 8.2.5 Spring 实现的搜索界面

通过 spring 应用程序上下文配置 HttpSolrServer 最简单的方式是定义一个 SolrServer 的



bean。然后把它注入到想要使用的类中。Spring 配置文件例子如下：

```
<bean id="solrServer"
      class="org.apache.solr.client.solrj.impl.HttpSolrServer">
  <constructor-arg>
    <value>http://localhost:8080/solr/core0</value>
  </constructor-arg>
</bean>
```

更完整的设置 solrServer 参数的例子：

```
<bean id="solrServer" class="org.apache.solr.client.solrj.impl.
HttpSolrServer">
  <constructor-arg value="${solr.serverUrl}"/>
  <property name="connectionTimeout" value="${solr.connectionTimeout}"/>
  <property name="defaultMaxConnectionsPerHost" value="${solr.defaultMax
ConnectionsPerHost}"/>
  <property name="maxTotalConnections" value="${solr.maxTotal
Connections}"/>
</bean>
```

在服务层使用这个叫做“solrServer”的 Spring bean。这里使用 spring IOC 机制，创建 org.apache.solr.client.solrj.SolrServer 对象作为服务中的一个成员变量。使用如下的代码和 Solr 打交道：

查询关键词：

```
SolrQuery query = new SolrQuery();
query.setQuery(search); //查询词是 search
QueryResponse qr = solrServer.query(query);
return qr.getBeans(SearchItem.class);
```

提交项目：

```
solrServer.addBean(item);
solrServer.commit();
```

删除项目：

```
solrServer.deleteById(id);
solrServer.commit();
```

也可以使用 Spring Data Solr 项目。这个项目的地址是：<https://github.com/SpringSource/>

spring-data-solr/。Spring Data Solr 是 Spring Data 的子项目。Spring Data Solr 实现了 Spring Data 访问 Solr 存储并提供了 Spring Data JPA(Java 持久化 API)模型的访问方式。此外, Spring Data Solr 提供了一个更底层的 SolrTemplate 以方便启动嵌入式的 Solr 服务器。SolrTemplate 是 Solr 操作的核心支持类。

Spring Data 提供了一套数据访问层(DAO)的解决方案, 致力于减少数据访问层的开发量。它使用一个叫作 Repository 的接口类为基础。Repository 定义如下:

```
public interface Repository<T, ID extends Serializable> {
}
```

Repository 是访问底层数据模型的超级接口。而对于某种具体的数据访问操作, 则在其子接口中定义。例如, Spring Data Solr 项目中定义了访问 Solr 中数据的 SolrRepository 接口。

所有继承 Repository 接口的界面都由 Spring 管理, 此接口作为标识接口, 功能就是用来控制领域模型。Spring Data 可以让我们只定义接口, 只要遵循 Spring Data 的规范, 就无需写实现类。Spring 可以根据接口中定义的方法名实现 Repository。

用一个简单的实例说明如何使用 Spring Data Solr。创建一个叫做 HttpSolrContext 的类并用@Configuration 注解标注这个类。用@EnableSolrRepositories 注解标注这个类来启用 Spring Data Solr 存储, 并且配置 Solr 存储的根包。

```
@EnableSolrRepositories("com.lietu.spring.datasolr.todo.repository.s
olr")
```

使用@PropertySource 注解标注这个类, 并把值设置成'classpath:application.properties'。用这个值配置属性文件的位置, 并且增加一个 PropertySource 到 Spring 的环境。

加一个 Environment 列到这个类, 并用@Resource 标注该列。注入的 Environment 用来取得加到属性文件的属性。

```
@Resource
private Environment environment;
```

创建一个叫做 solrServerFactoryBean()的方法, 并且用@Bean 注解标注这个方法。这个方法的实现创建一个新的 HttpSolrServerFactoryBean 对象, 然后设置 Solr server url 的值, 并返回这个创建的对象。



创建一个叫做 `solrTemplate()` 的方法，并且用 `@Bean` 注解标注这个方法。此方法的实现创建一个新的 `SolrTemplate` 对象，并传递使用过的 `SolrServer` 实现作为构造函数的参数。

```
@Bean
public SolrTemplate solrTemplate() throws Exception {
    return new SolrTemplate(solrServerFactoryBean().getObject());
}
```

`HttpSolrContext` 完整的源代码如下：

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.core.env.Environment;
import org.springframework.data.solr.core.SolrTemplate;
import
org.springframework.data.solr.repository.config.EnableSolrRepositories;
import
org.springframework.data.solr.server.support.HttpSolrServerFactoryBean;

import javax.annotation.Resource;

private Environment environment;

@Configuration
@EnableSolrRepositories("com.lietu.spring.datasolr.todo.repository.s
olr")
@PropertySource("classpath:application.properties")
public class HttpSolrContext {

    @Resource
    private Environment environment;

    @Bean
    public HttpSolrServerFactoryBean solrServerFactoryBean() {
        HttpSolrServerFactoryBean factory = new
HttpSolrServerFactoryBean();

        factory.setUrl(environment.getRequiredProperty("solr.server.url"));

        return factory;
    }

    @Bean
    public SolrTemplate solrTemplate() throws Exception {
```

```

        return new SolrTemplate(solrServerFactoryBean().getObject());
    }
}

```

通过如下步骤为 HTTP Solr 服务器创建一个 XML 配置文件：

- 通过使用上下文命名空间的属性占位符元素配置使用的属性文件。
- 启用 Solr 的存储库，并使用 solr 命名空间的存储元素配置 Solr 存储库的基础包。
- 利用 SOLR 命名空间的 solr-server 元素配置 HTTP Solr 服务器 bean。设置 Solr 服务器的 url。
- 配置 Solr 模板 bean。设置配置好的 HTTP Solr 服务器 bean 作为构造函数的参数。

exampleApplicationContext-solr.xml 文件内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:solr="http://www.springframework.org/schema/data/solr"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.2.xsd
http://www.springframework.org/schema/data/solr
http://www.springframework.org/schema/data/solr/spring-solr.xsd">

    <context:property-placeholder
location="classpath:application.properties"/>

    <!-- Enable Solr repositories and configure repository base package -->
    <solr:repositories
base-package="com.lietu.spring.datasolr.todo.repository.solr"/>

    <!-- Bean definitions -->
    <beans>
        <!-- Configures HTTP Solr server -->
        <solr:solr-server id="solrServer" url="${solr.server.url}"/>

        <!-- Configures Solr template -->
        <bean id="solrTemplate" class="org.springframework.data.solr.
core.SolrTemplate">
            <constructor-arg index="0" ref="solrServer"/>

```



```
</bean>
</beans>
</beans>
```

为了把文档加入到 Solr 索引，接下来创建一个文档类。文档类基本上是一个按照如下规则实现的 POJO：

- @Field 注解用于在 POJO 列和 Solr 文档列之间建立一个链接。
- 如果 bean 的列名称不等于文档的列名称，则文档列的名称必须作为 @Field 注解的值给出。
- 可以用 @Field 注解标注一个字段或 setter 方法。

Spring Data Solr 假设文档默认 id 字段的名称是 ‘id’。可以使用 @Id 注解标注 id 字段来覆盖此设置。创建一个叫做 TodoDocument 的类来索引要完成的任务条目。添加 id 字段到 TodoDocument 类，并用 @Field 注解标注该列。添加 description 字段到 TodoDocument 类，并用 @Field 注解标注该列。添加 title 字段到 TodoDocument 类，并用 @Field 注解标注该列。给 TodoDocument 类的字段创建 getter 方法。

创建一个叫做 Builder 的静态内部类用来构建新的 TodoDocument 对象。添加一个静态 getBuilder() 方法到 TodoDocument 类。此方法的实现返回一个新的 TodoDocument.Builder 对象。

TodoDocument 类的源代码如下所示：

```
import org.apache.solr.client.solrj.beans.Field;
import org.springframework.data.annotation.Id;

public class TodoDocument {

    @Id
    @Field
    private String id;

    @Field
    private String description;

    @Field
    private String title;

    public TodoDocument() {
```

```

    }

    public static Builder getBuilder(Long id, String title) {
        return new Builder(id, title);
    }

    //忽略了 Getters 方法

    public static class Builder {
        private TodoDocument build;

        public Builder(Long id, String title) {
            build = new TodoDocument();
            build.id = id.toString();
            build.title = title;
        }

        public Builder description(String description) {
            build.description = description;
            return this;
        }

        public TodoDocument build() {
            return build;
        }
    }
}

```

接下来创建存储接口。Spring Data Solr 存储的基本接口是 `SolrCrudRepository<T, ID>` 接口，并且每个存储接口必须扩展这个接口。

当扩展 `SolrCrudRepository<T, ID>` 接口时，必须给出 `T` 和 `ID` 两个类型参数。其中，`T` 类型参数表示文档类的类型。`ID` 类型参数表示文档 `id` 的类型。

可以按照以下步骤创建存储接口：

- 创建一个叫做 `TodoDocumentRepository` 的接口。
- 扩展 `SolrCrudRepository` 接口，并用文档类类型和它的 `ID` 类型(字符串)作为类型参数。

`TodoDocumentRepository` 接口的源代码如下所示：





```
import org.springframework.data.solr.repository.SolrCrudRepository;

public interface TodoDocumentRepository
    extends SolrCrudRepository<TodoDocument, String> {

}
```

下一步是使用创建的 Solr 存储来创建服务。首先创建一个叫做 `TodoIndexService` 的服务接口，然后实现这个接口。`TodoIndexService` 接口的源代码如下所示：

```
public interface TodoIndexService {

    public void addToIndex(Todo todoEntry); //增加文档到索引

    public void deleteFromIndex(Long id); //从索引删除文档

}
```

接下来，实现创建出来的接口。通过以下步骤实现服务接口：

- 创建一个服务类的框架实现。
- 实现把文档添加到 Solr 索引的方法。
- 实现从 Solr 的索引中删除文档的方法。

接下来详细讲解如何创建一个服务类的框架实现。

创建一个叫做 `RepositoryTodoIndexService` 的类，并用 `@Service` 注解标注这个类。这个注解把这个类标记成为一个服务，并确保在类路径扫描中检测到这个类。

增加一个 `TodoDocumentRepository` 字段到 `RepositoryTodoIndexService` 类，并用 `@Resource` 注解标注这个字段。这个注解指示 Spring IoC 容器把实际的存储实现注入到服务的存储列。

虚拟服务实现的源代码如下所示：

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.Resource;

@Service
public class RepositoryTodoIndexService implements TodoIndexService {
```

```

@Resource
private TodoDocumentRepository repository;

//在这里添加方法
}

```

创建把文档添加到 Solr 索引的方法。

在 `Repository_TODO_IndexService` 类中添加 `addToIndex()` 方法，然后使用 `@Transactional` 注解标记这个方法。这将确保 Spring Data Solr 参与 Spring 的事务管理。通过调用 `TodoDocumentRepository` 接口的 `save()` 方法添加文档到 Solr 索引。创建方法的源代码如下所示：

```

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.Resource;

@Service
public class Repository_TODO_IndexService implements TODO_IndexService {

    @Resource
    private TodoDocumentRepository repository;

    @Transactional
    @Override
    public void addToIndex(TODO todoEntry) {
        TodoDocument document = TodoDocument.getBuilder(todoEntry.
getId(),
                todoEntry.getTitle())
                .description(todoEntry.getDescription())
                .build();

        repository.save(document);
    }

    //在这里添加 deleteFromIndex() 方法
}

```

使用如下步骤创建从 Solr 索引删除文档的方法：

- 添加 `deleteFromIndex()` 方法到 `Repository_TODO_IndexService` 类，并用 `@Transactional` 注解标记此方法。这将确保 Spring Data Solr 存储参与 Spring 的事务管理。



- 通过调用 `TodoDocumentRepository` 接口的 `delete()` 方法从 Solr 的索引中删除文件。

创建的方法源代码如下所示：

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.Resource;

@Service
public class Repository_TODO_IndexService implements TODO_IndexService {

    @Resource
    private TodoDocumentRepository repository;

    //在这里添加 addToIndex() 方法

    @Transactional
    @Override
    public void deleteFromIndex(Long id) {
        repository.delete(id.toString());
    }
}
```

通过使用查询方法实现搜索功能。可以使用以下技术创建 Spring Data Solr 的查询方法：

- 根据方法名生成查询；
- 命名查询；
- `@Query` 注解。

这三种方法中，根据方法名生成查询是最简单的一种方法。这里先只介绍这种方法，其它方法可以察看 Spring Data Solr 的说明文档。从方法名生成查询是一个查询生成策略，从查询方法的名称解析出来要执行的查询。

查询方法的名称必须用一个特殊的前缀标识查询方法。这些前缀是：`find`, `findBy`, `get`, `getBy`, `read` 和 `readBy`。当解析要执行的查询时，从方法名称中剥离出这个前缀。

用属性表达式指出文档类的属性。可以组合多个属性表达式，方法是：在它们之间加入 `And` 或 `Or` 关键字。

查询方法的参数数量必须与在名称中使用的属性表达式的数量相等。例如，

TodoDocumentRepository 接口的源代码中的 findByTitleContainsOrDescriptionContains 方法包括 title 和 description 两个参数。

TodoDocumentRepository 接口的源代码如下所示：

```
import org.springframework.data.solr.repository.SolrCrudRepository;
import java.util.List;

public interface TodoDocumentRepository
    extends SolrCrudRepository<TodoDocument, String> {

    public List<TodoDocument> findByTitleContainsOrDescriptionContains
        (String title,
                                                String description);
}
```

**注意：**如果搜索词包含多个单词，那么这个查询方法就行不通了。

使用已经创建好的查询方法。首先在 TodoIndexService 接口中声明 search()方法，然后添加 search()方法的实现到 RepositoryTodoIndexService 类。

调用 TodoDocumentRepository 接口的 findByTitleContainsOrDescriptionContains()方法返回一个 TodoDocument 对象组成的列表。

RepositoryTodoIndexService 类的相关部分如下所示：

```
import org.springframework.stereotype.Service;
import javax.annotation.Resource;
import java.util.List;

@Service
public class RepositoryTodoIndexService implements TodoIndexService {

    @Resource
    private TodoDocumentRepository repository;

    @Override
    public List<TodoDocument> search(String searchTerm) {
        return repository.findByTitleContainsOrDescriptionContains
            (searchTerm, searchTerm);
    }
}
```



### 8.2.6 Solr 索引库的查找

通过 Solr 的管理界面，可以直接分析索引库。还可以直接在浏览器输入 URL 地址来查询索引库。通过各种参数来指定搜索执行的方式和返回结果的格式等。Solr 中的参数分成基本参数和公共参数等类型。

Solr 中所有请求都会用到的核心查询参数说明如下。

- qt——查询类型（request handler），例如 standard。
- wt——返回格式类型（response writer），例如 XML 或 JSON。

公共的查询参数有：

- q——查询词；
- sort——排序方式；
- start——返回结果的开始行；
- rows——本次需要返回结果的行数；
- fl——需要返回的列名称。

例如，执行一个最基本的关键词搜索，搜索“NBA”这个词：

```
http://localhost:8080/solr/select/?q=NBA&version=2.2&start=0&rows=10
&indent=on
```

这里返回最开始的前 10 行搜索结果。

执行分类统计搜索。假设 cat 列是分类列，首先我们可以看一下这个索引分了几类。

```
http://localhost/solr/select/?q=%3A*&rows=0&facet.field=cat&rows=0&
facet=true&&facet.limit=-1
```

部分返回结果如下所示：

```
<result name="response" numFound="1937323" start="0"/>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="cat">
      <int name="产品">395413</int>
```

```

<int name="供求">151571</int>
<int name="其他">86733</int>
<int name="厂商">175977</int>
<int name="合作">75374</int>
<int name="图书">8066</int>
<int name="培训">6053</int>
<int name="展会">10761</int>
<int name="技术文摘">75885</int>
<int name="教程">8516</int>
<int name="新闻">150630</int>
<int name="方案及应用">14282</int>
<int name="特价">360621</int>
<int name="研究">139945</int>
<int name="职位">10565</int>
<int name="论文">265031</int>
<int name="销售">1900</int>
</lst>
</lst>
<lst name="facet_dates"/>
</lst>
...

```

如果要按 URL 地址查找一个网站的数据，则需要把 URL 地址中的字符“:”和“/”转义：

```
url:http\:\//2packaging*
```

可以通过 `qt` 参数来调用不同的 `RequestHandler` 处理查询请求，下面这个查询调用 `StandardRequestHandler` 来处理。

```
http://localhost:8983/solr/select/?q=video&fl=name+score&qt=standard
```

**Solr** 的分析页面是不可缺少的实验方法和故障排除工具。可以用这个工具分析不同的语句来验证是否是期望的结果。如果查询结果和你所想要的结果不同，也可以利用这个故障排除工具找出原因。在 **Solr** 的管理页面顶部有一个[ANALYSIS]的超级链接。

在图 8-5 这个页面顶部的第一个选项表示字段的类型，这一项是必选的。既可以直接输入字段类型，也可以按字段名称来分析。但这个工具是用来分析文本型字段类型的，而不是用来分析布尔类型、日期类型和数字类型的。这个工具能同时分析索引和/或查询文本。如果同时做查询和索引分析并且想查看分析时索引部分的匹配情况，可以高亮显示匹配结果。

输入 “Quoting,Wi-Fi” and stopword 后，分析的输出结果如图 8-6 所示。

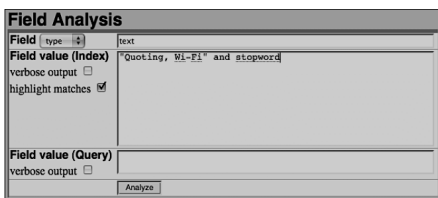


图 8-5 分析英文的 Web 界面

org.apache.solr.analysis.WordDelimiterFilterFactory (catenateWords=1, catenateNumbers=1, splitOnCaseChange=1, catenateAll=0, generateNumberParts=1, generateWordParts=1)

term position	1	2	3	5
term text	Quoting	Wi	Fi	stopword
term type	word	word	word	word
source start,end	1,8	10,12	13,15	22,30
payload				

图 8-6 分析英文的结果输出

输出结果中最重要的行是叫 “term text” 的第二行。term 是实际存储和查询的原子单位。因此，查询的分析结果必须和索引分析阶段的结果包含相同的 term 才能够匹配。请注意，位置 3 上有两个词条。在同一位置的多个词条可能是由于同义词扩展产生，但在这里，是由 WordDelimiterFilterFactory 引入了 WiFi 这个词。Quoting 在词干化和小写化之后变成 quot。因为 and 在停用词表里所以被 StopFilter 省略。

使用 StopFilter 的副作用是，因为有些查询中的词全部在停用词表中，所以可能完全搜索不到了。如果不想直接去掉停用词，可以用 CommonGramsFilter 来改进短语查询。CommonGramsFilter 对布尔查询没有意义。布尔查询仅仅通过 CommonGramsFilter 而不发生改变。

有两个相关的类：CommonGramsFilter 和 CommonGramsQueryFilter。在索引时使用 CommonGramsFilter，在查询时使用 CommonGramsQueryFilter。CommonGramsFilter 输出 CommonGrams 和 Unigrams，这样就可以使用布尔查询代替短语查询。

下面是一个使用 CommonGramsFilter 的 schema.xml 配置文件例子：

```
<fieldType name="CommonGramTest" class="solr.TextField" position
IncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="ISOLatin1AccentFilterFactory"/>
    <filter class="solr.PunctuationFilterFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.CommonGramsFilterFactory" words="new400common.
txt"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="ISOLatin1AccentFilterFactory"/>
```

```

<filter class="solr.PunctuationFilterFactory"/>
<filter class="solr.LowerCaseFilterFactory"/>
<filter class="solr.CommonGramsQueryFilterFactory" words="
new400common.txt"/>
</analyzer>
</fieldType>

```

有没有办法把最满足条件的文档排在最前面，如果查找多个关键词时，如何把都满足的或者满足得最多的结果排在最前面？

可以制定如下规则：如果有三个条件，那么这三个条件都必须满足，如果条件超过三个，只要满足 75% 的条件即可。按照这个思路，首先重新配置 `solrconfig.xml`，修改 `dismax` 的默认参数配置，否则有些列名可能无效。

```

<requestHandler name="dismax" class="solr.DisMaxRequestHandler" >
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <float name="tie">0.01</float>
    <str name="qf">
      body^0.5 title^1.2
    </str>
    <str name="pf">
      body^0.2 title^1.5
    </str>
    <str name="fl">
      title,body
    </str>
    <str name="mm">
      75%
    </str>
    <int name="ps">100</int>
    <str name="q.alt">*:*</str>
    <!-- example highlighter config, enable per-query with hl=true -->
    <str name="hl.fl">text features name</str>
    <!-- for this field, we want no fragmenting, just highlighting -->
    <str name="f.name.hl.fragsize">0</str>
    <!-- instructs Solr to return the field itself if no query terms are
      found -->
    <str name="f.name.hl.alternateField">name</str>
    <str name="f.text.hl.fragmenter">regex</str> <!-- defined below -->
  </lst>
</requestHandler>

```





然后在浏览器输入：

```
http://localhost/solr/select/?q=test&rows=0&qt=dismax&mm=75%
```

这里设置了查询满足条件“mm=75%”并且指定 Solr 的 `DisMaxRequestHandler` 类来处理查询请求。

### 8.2.7 索引分发

为了支持更多的搜索访问量或者避免单点失败，使索引服务有更高的可用性，需要分发索引到多个服务器上同时提供服务。利用 `rsync` 压入和弹出索引快照不会对服务器性能有太大的影响。下面介绍如果有一个已经存在的 Lucene 索引，如何把这个索引从主节点复制到从节点。

在每个 Linux 服务器上创建路径：

```
/usr/local/solr/
```

这个路径下包含一些重要的可执行脚本：

主节点需要的脚本如下：

- `rsyncd-enable`
- `rsyncd-disable`
- `rsyncd-start`
- `rsyncd-stop`
- `snapshotter`
- `snapcleaner`

从节点需要的脚本如下：

- `snappuller-enable`
- `snappuller-disable`
- `snappuller`
- `snapinstaller`
- `snapcleaner`

在 Solr 文件夹下创建一个 `conf` 目录，`conf` 目录下放一个配置文件：`scripts.conf`。

主节点中 `scripts.conf` 的配置文件如下所示：

```
solr_hostname=localhost
solr_port=8983
rsyncd_port=18983
data_dir=<lucene 索引路径>
webapp_name=solr
master_host=
master_data_dir=
master_status_dir=
```

**注意：**如果 `index` 的目录路径为 `/usr/local/lucene/search/index/`，则配置文件里 `data_dir` 路径应该为：`/usr/local/lucene/search/`。

从节点中 `scripts.conf` 的配置文件如下所示：

```
user=
solr_hostname=<主节点 ip 地址>
solr_port=8993
rsyncd_port=18993
data_dir=<从节点的 lucene 索引路径>
webapp_name=solr
master_host=<主节点 ip 地址>
master_data_dir=<主节点的 lucene 索引路径>
master_status_dir=<主节点的 solr 路径>/logs/clients/
```

例如这里 `master_status_dir` 可能是 `/usr/local/solr/logs/clients/`。

通过如下步骤即可开始复制索引。

① 在主节点里，运行 `solr/bin` 目录下面的 Linux 命令：

```
#rsyncd-enable
#rsyncd-start
```

执行这两个命令后，在 `solr/logs` 目录下将会创建一个 `pid` 文件和一个 `log` 文件。

② 为了做一个索引的快照，只需要在主节点运行以下命令即可：

```
#snapshotter
```

现在可以在 `crontab` 里进行配置，`snapshotter` 命令在一天的某几个时间运行。当新创建



一个快照时，索引必需是完整的，也就意味着如果正在写入索引时，则不能运行 `snapshotter` 命令。

如果运行 `snapshotter` 时，增加 `-c` 参数，只有当最后一个快照改变时，才创建一个新的 `snapshotter`。为了防止磁盘被快照塞满，可以经常运行 `snapcleaner` 命令。运行 `snapshotter -N 2` 将会移除所有旧的快照，只保留两个最新的快照。

③ 在从节点，需要执行如下的脚本拉回并且安装快照：

```
#snappuller-enable  
#snappuller -P 18983  
#snapinstaller
```

注意：`snappuller` 后面的端口号是在主节点配置的 `rsyncd_port`，同时在从节点应该调度执行 `snapcleaner`。

④ 现在在从节点调度执行 `snappuller` 和 `snapinstaller` 的问题就剩下到主节点里做身份认证了。为了解决这个问题，可以添加 `ssh-keys` 到主节点的认证文件中。

在从节点执行以下命令：

```
#ssh-keygen -t dsa
```

执行 `ssh-keygen` 命令后，认证文件生成到 `home` 目录下的 `.ssh` 目录中。

用 `scp` 命令把从节点上的 `id_dsa.pub` 文件复制到主节点：

```
#scp id_dsa.pub username@masternode:./id_dsa.pub
```

在主节点的 `.ssh` 目录执行以下命令：

```
#touch authorized_keys  
#chmod 600 authorized_keys  
#cat ../id_dsa.pub >> authorized_keys
```

## 8.2.8 Solr 搜索优化

如果要在做索引时给不同的列设置不同的权重：

```
<field name="myBoostedField" boost="7.0">value1</field>  
<field name="myBoostedField" boost="8.0">value2</field>
```

```
<field name="myBoostedField" boost="4.0">value3</field>
```

发送数据的写法:

```
XML.writeXML(xmlContent, "field", value1, "name",
              " myBoostedField","boost","7.0");
```

也可以在搜索的时候动态加权。对于标准的请求处理器，对标题列加权:

```
q=title:superman^2 subject:superman。
```

使用 **dismax** 请求处理器，可以在参数 **qf** 中对指定的列申明加权，例如:

```
q=superman&qf=title^2 subject。
```

使用函数式查询实现日期和相关度混合排序:

```
queryWord += " AND_val_:\\"linear( recip(rord(timestamp),1,10000,10000),
10000000,0)\\\"";
```

在索引中出现“头疼 药”和“头疼药”这样的内容，其分词都是“头疼”和“药”，前者有搜索结果，后者无搜索结果。这样的原因是 Solr 的查询词解析类使用的是短语匹配的方式。“头疼药”只按照“短语匹配”搜索是不恰当的，应该是先短语匹配，后按照分词进行垂直搜索。为了实现这样的效果，修改 **SolrQueryParser** 类。

```
protected Query getFieldQuery(String field, String queryText) throws
ParseException {
    //如果碰到“-”，则把查询当成内部函数处理。
    if (field.equals("_val_")) {
        return QueryParsing.parseFunction(queryText, schema);
    }
    //默认执行普通的字段查询
    TokenStream source = this.getAnalyzer().tokenStream(field, new
    StringReader(queryText));
    ArrayList<Token> v = new ArrayList<Token>(10);
    Token t;
    while (true) {
        try {
            t = source.next();
        }
        catch (IOException e) {
            t = null;
        }
        if (t == null)
```



```
        break;
    v.add(t);
}
try {
    source.close();
}
catch (IOException e) {
    //省略
}
if (v.size() == 0)
    return null;
else if (v.size() == 1)
    return new TermQuery(new Term(field, ((Token)v.get(0)).termText()));
else {
    PhraseQuery q = new PhraseQuery();
    BooleanQuery b = new BooleanQuery();
    q.setBoost(2048.0f);
    b.setBoost(0.001f);
    for (int i = 0; i < v.size(); i++) {
        Token token = v.get(i);
        q.add(new Term(field, token.termText()));
        TermQuery tmp = new TermQuery(new Term(field, token.termText()));
        tmp.setBoost(0.01f);
        b.add(tmp, BooleanClause.Occur.MUST);
    }
    BooleanQuery bQuery = new BooleanQuery();
    //用 OR 条件合并两个查询
    bQuery.add(q, BooleanClause.Occur.SHOULD);
    bQuery.add(b, BooleanClause.Occur.SHOULD);
    return bQuery;
}
}
```

如果需要限制这里的 BooleanQuery bQuery 的查询范围，还可以加入下列代码：

```
if(!stopwords.contains(token.termText())){
    b.add(tmp, BooleanClause.Occur.MUST); //OR 连接
    System.out.println("add tmp:"+tmp);
}
```

在通常的搜索中用户可以按内容相关度排序，或者按照日期逆序排序。

```
queryWord; createtime desc
```

还可以按 sort 参数排序：

```
inStock desc, price asc
```

SolrJ 中按时间列排序的例子如下所示：

```
query.addSortField("timestamp", ORDER.desc);
```

经常通过一个选项来切换这两种排序方式。为了更加简化搜索界面，可以综合内容相关度排序和日期排序。基本的方法是设置时间加权，在考虑到搜索词相关性的同时考虑到搜索结果的时间。例如，索引库中有三个字段：标题、内容、日期。组合搜索“内容+标题”，然后按时间排序。这样有个问题：比如我现在搜索“手机”，结果中如果不按日期排序，那结果的相关性很好，但是如果按日期排序，有些商情的内容里面会有联系人的手机号这种字样，但是其实这条商情并不是卖手机的，有可能是卖衣服的。但是这条商情的权重还是会排到卖手机的上面。Solr 通过函数查询来实现时间加权排序。

- OrdFieldSource 实现了 ord(myfield)函数。
- ReverseOrdFieldSource 实现了 rord(myfield)函数。
- LinearFloatFunction 实现了数值列的 linear(myfield,1,2) 函数。
- MaxFloatFunction 实现了数值列或常量的 max(linear(myfield,1,2),100) 函数。
- ReciprocalFloatFunction 实现了数值列的 recip(myfield,1,2,3) 函数。

实际搜索“NBA”这个词的时候，使用时间加权的例子如下所示：

```
+_val_:"linear\ (recip\ (rord\ (timestamp\),1,10000,10000\),10000000,0\)" NBA
```

其基本原理是用索引中“timestamp”列的值来影响排序结果。

在优化索引阶段，Solr 需要大约 2 倍索引大小的临时空间，因此需要大约 400 GB 的硬盘来容纳 200 GB 的索引。



## 8.3 Solr 扩展与定制

Solr 的实现很灵活，可以定制输入输出或者增加内部功能。

### 8.3.1 Solr 中字词混合索引

在 Lucene 的介绍中已经提到了实现字词混合索引的方法。在此基础上增加



FilterFactory。

```
public class SingleFilterFactory extends BaseTokenFilterFactory {
    public SingleFilter create(TokenStream input) {
        return new SingleFilter(input);
    }
}
```

在 schema.xml 中定义 text 列类型如下所示：

```
<fieldType name="text" class="solr.TextField" positionIncrementGap="100">
    <analyzer type="index">
        <tokenizer class="CnTokenizerFactory"/>
        <filter class="solr.SingleFilterFactory"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="CnTokenizerFactory"/>
        <filter class="solr.SingleFilterFactory"/>
    </analyzer>
</fieldType>
```

修改 org.apache.solr.search 包中的 SolrQueryParser 类：

```
//默认执行普通的字段查询
TokenStream source = this.getAnalyzer().tokenStream(field,
    new StringReader(queryText));

ArrayList<Token> v = new ArrayList<Token>(10);
Token t;
while (true) {
    try {
        t = source.next();
    }
    catch (IOException e) {
        t = null;
    }
    if (t == null)
        break;
    v.add(t);
}
try {
    source.close();
}
catch (IOException e) {
```

```

        //省略
    }

    if (v.size() == 0)
        return null;
    else if (v.size() == 1)
        return new TermQuery(new Term(field, ((Token)v.get(0)).termText()));
    else {
        PhraseQuery q = new PhraseQuery();
        q.setBoost(2048.0f);
        ArrayList<SpanQuery>s=new ArrayList<SpanQuery>(v.size());
        for (int i = 0; i < v.size(); i++) {
            Token token = v.get(i);
            if(token.getPositionIncrement()>0) {
                q.add(new Term(field, token.termText()));
            }
            if(token.termText().length()==1) {
                SpanTermQuery tmp =
                    new TermQuery(new Term(field, token.termText()));
                s.add(tmp);
            }
        }

        BooleanQuery bQuery = new BooleanQuery();
        //用 OR 条件合并两个查询
        bQuery.add(q, BooleanClause.Occur.SHOULD);
        if(s.size()>0) {
            SpanNearQuery nearQuery =
                new SpanNearQuery(s.toArray(new SpanQuery[s.size()]),
                    s.size(),true);
            nearQuery.setBoost(0.001f);
            bQuery.add(nearQuery, BooleanClause.Occur.SHOULD);
        }
        return bQuery;
    }
}

```

### 8.3.2 相关检索

经常需要实现相似检索的功能。例如输入一篇文章，返回相关的 5 篇文章。Solr 中包括了一个 MoreLikeThis 的处理模块。在 solrconfig.xml 中包括 MoreLikeThisHandler 的定义：

```

<requestHandler name="/mlt" class="solr.MoreLikeThisHandler">
  <lst name="defaults">
    <str name="mlt.fl">title,abstract</str>

```



```

        <int name="mlt.mindf">1</int>
    </lst>
</requestHandler>

```

mlt.fl 中包括提取关键词的默认列。

搜索的时候输入下面的内容：

```

http://www.lietu.com:8080/solr/select/?q=%E6%B1%BD%E8%BD%A6&version=2.2
&start=0&rows=1&mlt=true&mlt.mindf=1&mlt.mintf=1&mlt.fl=title,abstract

```

返回的结果后面包括了相似匹配的结果：

```

<lst name="moreLikeThis">
    <result name="521838" numFound="6586103" start="0">
    <doc>
        <str name="abstract">

```

本文通过分析世界汽车工业的发展趋势，总结出 21 世纪世界汽车工业发展的八大趋势，主要表现在汽车产业组织、汽车生产方式、汽车产品及造型和汽车可持续发展战略上。

```

        </str>
    <str name="cn_institution">武汉理工大学</str>
    <str name="en_abstract"/>
    <str name="en_title">
    Eight Developmont Trends of The World Automobile Industry in The
    21st Century
    </str>
    <str name="id">5141078</str>
    <str name="pin_yin_name">杨瑞海, 韩雄辉</str>
    <str name="title">21 世纪世界汽车工业发展的八大趋势</str>
    <str name="user_real_name">杨瑞海, 韩雄辉</str>
    </doc>
    ...
</result>
</lst>

```

这个默认的相关检索仍然有待改进，因为 MoreLikeThis 查询的准确性依赖于提取关键词的准确性，为了提取关键词，首先要做的工作是去掉 StopWord。MoreLikeThis 类默认使用的是英文的 StopWord。下面我们修改 MoreLikeThisHandler 让它从外部读取 StopWord.txt。

```

<requestHandler name="/mlt" class="solr.MoreLikeThisHandler">
    <lst name="defaults">

```

```

        <str name="mlt.fl">manu,cat</str>
        <int name="mlt.mindf">1</int>
    </lst>
    <str name="stopWordFile">stopwords.txt</str>
</requestHandler>

```

MoreLikeThisHandler 类的实现代码修改如下：

```

private static Set stopWords = null;

public void init(NamedList args) {
    super.init(args);
    SolrParams p = SolrParams.toSolrParams(args);
    String stopWordFile = p.get("stopWordFile");
    if(stopWordFile == null) {
        stopWords = StopFilter.makeStopSet(StandardAnalyzer.STOP_WORDS);
    }
    return;
}

List<String> wlist;
try {
    wlist = Config.getLines(stopWordFile);
} catch (IOException e) {
    throw new SolrException( SolrException.ErrorCode.NOT_FOUND,
        "MoreLikeThis requires a stop word list " );
}

stopWords = StopFilter.makeStopSet( (String[])wlist.toArray(new
String[0]), true);
}
...
mlt.setStopWords(stopWords);
...

```

### 8.3.3 搜索结果去重

折叠对于一个给定列的相同或相似值的搜索结果叫做 **collapsing**。例如对同一站点搜索结果的折叠，经常也会在这个搜索结果上加上“...中还有几条相关信息”，如图 8-7 所示。

这需要我们给 Solr 增加折叠的功能。首先我们假定折叠的列是未分词的。

<https://issues.apache.org/jira/browse/SOLR-236> 正是关于这个问题的解决方法。先取得这个版本：

```

# svn export -r592129 http://svn.apache.org/repos/asf/lucene/solr/
# patch -u -p0 < field-collapsing-extended-592129.patch

```

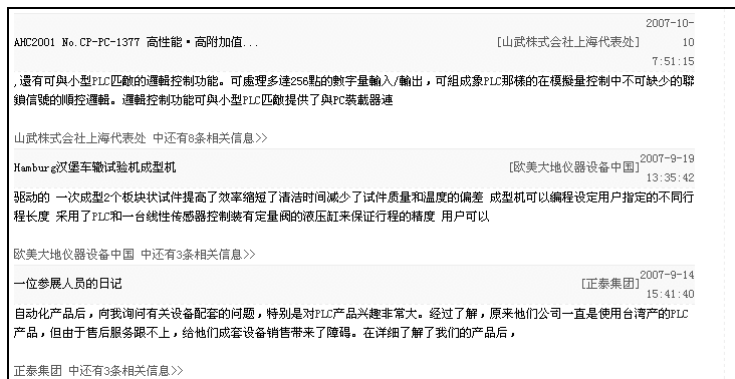


图 8-7 搜索结果去重效果图

搜索测试：

```
http://localhost:8080/select/?q=words_t%3AApple&version=2.2&start=0&rows=10&indent=on&collapse.field=t_s&collapse.threshold=1
```

返回的结果如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
  <response>

    <lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">0</int>
      <lst name="params">
        <str name="start">0</str>
        <str name="collapse.max">1</str>
        <str name="indent">on</str>
        <str name="q">words_t:Apple</str>
        <str name="version">2.2</str>
        <str name="rows">10</str>
        <str name="collapse.field">t_s</str>
      </lst>
    </lst>

    <result name="response" numFound="2" start="0">
      <doc>

        <int name="c_i">3</int>
        <str name="id">1</str>
        <int name="popularity">0</int>
        <str name="sku">1</str>
        <str name="t_s">movie</str>
```

```

<date name="timestamp">2007-12-21T15:22:42.211Z</date>
<str name="words_t">Apple Orange</str>
</doc>
<doc>
<int name="c_i">4</int>
<str name="id">3</str>
<int name="popularity">0</int>
<str name="sku">3</str>
<str name="t_s">book</str>
<date name="timestamp">2007-12-22T02:01:53.328Z</date>
<str name="words_t">Apple Orange</str>
</doc>
</result>
<lst name="collapse_counts">
<str name="field">t_s</str>
<lst name="doc">
<int name="1">1</int>
</lst>
<lst name="count">
<int name="movie">1</int>
</lst>
<str name="debug">HashDocSet(1) Time(ms): 0/0/0/0</str>
</lst>
</response>

```

这样的返回结果表示：对于列“t\_s”，还有一个同名列“t\_s”的值是 movie。

其中主要参数介绍如下。

- collapse.facet = before|after 参数控制 faceting 是在 collapsing 前或后发生。
- collapse.threshold 参数控制在多少重复后开始折叠结果。
- collapse.maxdocs 参数控制折叠执行时最多考虑的文档数量。当结果集很大的时候，增加此参数能缩短执行时间。
- collapse.info.doc 和 collapse.info.count 参数控制 collapse\_counts 返回结果内容。

在实际应用场景中，设想一个 solr 索引库包含很多新闻故事，这些故事来源于许多报纸或有限电视。一条新闻故事可能来源于多个不同的报纸，例如《人民日报》或一些地方小报等。每个报纸对同一个新闻起上不同的标题，并截成不同的长度。需要检测并把这些重复的故事分组在一起显示。假设每个故事都由一个 Hash 整数代表（也就是语义指纹），例如取故事的几个关键词作为“similarity\_hash”，把这个值索引并存储起来作为检测重复



故事的依据。

我们可以基于这个“similarity\_hash”值来折叠搜索结果，这样同一个新闻故事的多次出现就折叠到了一起。而且，用户可能更愿意读到更加权威的版本，需要把这个结果优先显示到搜索结果中，当然也会有个重复新闻的计数和连接。权威性的取值可能是有限电视新闻网、国家级报纸、表示地区报纸、表示地方小报。可以把权威性的取值索引和存储成一个整数权值——authority。

然后，可以显示给用户以下结果：

“人咬狗”

\*\*日报，连接可见其他 77 个重复新闻

通过折叠 similarity\_hash 列实现这个功能，选择基于另外一列 authority 的值返回显示的新闻中的一条。

这样需要进一步修改这个折叠列的实现，增加一个参数：

```
collapse.authority=[field] //索引列，用来控制折叠后的组中返回哪一个值
```

以前 CollapseFilter 只对一列排序，然后在排序后的结果中发现重复，实现折叠功能。现在改为实现多个列排序，把 collapse.authority 也加到排序列中。我们修改它的主要实现 src/java/org/apache/solr/search/CollapseFilter.java 文件：

```
if (collapseType == CollapseType.NORMAL) {
    if(sort!=null) {
        SortField[] ofields = sort.getSort();
        SortField[] nfields = new SortField[ofields.length+1];
        for(int k=0;k<ofields.length;++k) {
            nfields[k] = ofields[k];
        }
        nfields[nfields.length-1] = new SortField(collapseField);
        sort.setSort(nfields);
    } else {
        sort = new Sort(new SortField(collapseField));
    }
}
```

### 8.3.4 定制输入输出

Solr 的输入和输出格式是固定的，有时候为了方便其他系统调用，需要把输入的 GET 请求改成 POST 请求。POST 请求可以用如下表单模拟：

```
<form action="Seach" method="post">
  <input type="text" name=" ers_word" />
  <input type="submit" value="submit" />
</form>
```

搜索结果需要返回的格式规定如表 8-3 所示。

表 8-3 返回结果格式

变量名称	变量命名	说 明
检索结果基本信息	baseinfo	ers_word、ers_id、ers_s、ers_tc、ers_nc 的父节点
检索词	ers_word	字符型字符串，最大长度 120 字节
检索 ID	ers_id	数字型字符串，最大长度 20 字节
检索状态	ers_s	字符型字符串
检索耗时	ers_tc	数字型字符串
检索结果数量	ers_nc	数字型字符串
检索结果内容	r	ers_n、ers_pn 父节点
检索结果顺序码	ers_n	数字型字符串
检索结果对应所在电子书中的页码	ers_pn	数字型字符串，非负

例如下面这个搜索结果：

```
<root>
  <baseinfo ers_word="软件" ers_id="123123" ers_s="000000" ers_tc="31"
    ers_nc="1299" />
  <r ers_n="1" ers_pn="">软件开发</r>
  <r ers_n="2" ers_pn="">懂 Protel 等软件，熟悉电子线路</r>
  <r ers_n="3" ers_pn="">一般办公室庶务，文件整理，进销存软件操作</r>
  <r ers_n="4" ers_pn="">计算机软件相关专业本科以上，英语 4 级以上；有视频会议
    系统、流媒体软件、网络通讯、图像处理、数据库等相关软件开发经验。</r>
</root>
```

在服务器端定义一个 Search 类（Servlet）来处理查询请求。

```
Public class Search {
  public void doPost(HttpServletRequest request, HttpServletResponse
    response)
```



```
        throws ServletException, IOException {
//1、接受用户查询请求参数
String ers_word = request.getParameter("ers_word");
if(ers_word==null||ers_word.length()<=0){
    response.sendError( 400, "检索词不能为空" );
    return ;
}

final SolrCore core = SolrCore.getSolrCore();
SolrServletRequest solrReq = new SolrServletRequest(core, request);

//2、将参数传递到 solr 内部
SolrParams spold = solrReq.getParams();
NamedList nl = spold.toNamedList();
nl.add("q", ers_word);
nl.add("start", 0);
nl.add("rows", 20);
Map <String, String []> map = spold.toMultiMap(nl);
SolrParams sp = new ServletSolrParams(map);
solrReq.setParams(sp);

//经过以上处理 solr 已经基本完成查询配置
SolrQueryResponse solrRsp = new SolrQueryResponse();
try {
    SolrRequestHandler handler
        = core.getRequestHandler(solrReq.getQueryType());
    if (handler==null) {
        log.warn("Unknown Request Handler '" + solrReq.getQueryType()
            + "' : " +
solrReq);
        throw new
SolrException(SolrException.ErrorCode.BAD_REQUEST,"Unknown Request
Handler '" +
solrReq.getQueryType() + "'", true);
    }

//3、Solr 执行查询
core.execute(handler, solrReq, solrRsp );
if (solrRsp.getException() == null) {

//4、调用自定义输出类进行输出
MyXMLWriter responseWriter = new MyXMLWriter();
response.setContentType(responseWriter.getContentType
(solrReq,solrRsp));
PrintWriter out = response.getWriter();
```

```

        responseWriter.rewrite(out, solrReq, solrRsp);

    } else {
        Exception e = solrRsp.getException();
        int rc=500;
        if (e instanceof SolrException) {
            rc=((SolrException)e).code();
        }
        sendErr(rc, SolrException.toString(e), request, response);
    }
} catch (SolrException e) {
    if (!e.logged) SolrException.log(log,e);
    sendErr(e.code(), SolrException.toString(e), request, response);
} catch (Throwable e) {
    SolrException.log(log,e);
    sendErr(500, SolrException.toString(e), request, response);
} finally {
    solrReq.close();
}
}
}

```

自定义 MyXMLWriter 和 MyWriter 类。MyXMLWriter 继承 XMLResponseWriter，而 MyWriter 继承 XMLWriter。在 Servlet 里调用 MyXMLWriter 输出搜索结果。

```

public class MyXMLWriter extends XMLResponseWriter {
    public void rewrite(Writer writer, SolrQueryRequest req, SolrQuery
        Response rsp) throws IOException {
        //MyXMLWriter 调用 MyWriter 进行输出
        MyWriter.rewriter(writer, req, rsp);
    }
}

public class MyWriter extends XMLWriter {
    public static float CURRENT_VERSION=2.2f;
    private static final char[] XML_START="<?xml version=\"1.0\"
        encoding=\"UTF-8\"?>\n".toCharArray();

    private static final char[] XML_START_NOSCHEMA=("<root>\n").toChar
        Array();

    public MyWriter(Writer writer, IndexSchema schema, SolrQueryRequest
        req,String version) {
        super(writer,schema,req,version);
    }
}

```





```
public static void rewriter(Writer writer, SolrQueryRequest req,
    SolrQueryResponse rsp) throws IOException{
    String ver = req.getParams().get(CommonParams.VERSION);

    MyWriter xw = new MyWriter(writer, req.getSchema(), req, ver);
    xw.defaultFieldList = rsp.getReturnFields();

    String ers_word = req.getParam("ers_word");
    //获取查询结果集。
    NamedList lst = rsp.getValues();

    StringBuilder sb = new StringBuilder("");
    //添加 XML 头信息。
    sb.append(XML_START).append(XML_START_NOSHEMA);

    //添加 XML 结果状态信息
    NamedList response = rsp.getResponseHeader();
    if(response!=null){
        String ers_s = (String)response.get("ers_s");
        String ers_tc = (String)response.get("ers_tc");
        String ers_nc = (String)response.get("ers_nc");

        XML.writeXML(sb, "beseinfo", "",
            "ers_word",ers_word,, "ers_s",ers_s,"ers_tc",ers_tc,"ers_nc",ers_nc);
    }
    //添加 XML 详细结果信息
    Object result = lst.get("result");
    if(result!=null){
        DocList docs = (DocList)result;
        SolrIndexSearcher searcher = req.getSearcher();
        DocIterator iterator = docs.iterator();
        int ss = docs.size();
        for (int i=0; i<ss; i++) {
            int id = iterator.nextDoc();
            Document doc = searcher.doc(id,defaultFieldList);
            Fieldable ers_pn = doc.getFieldable("ers_pn");
            Fieldable val = doc.getFieldable("describe");
            XML.writeXML(sb, "r", val==null ?"":val.stringValue(),
                "ers_n",i+1,"ers_pn",ers_pn==null?"":ers_pn.stringValue());
        }
    }
    //添加 XML 结尾
    sb.append("\n</root>\n");
}
```

```

        String temp = sb.toString();
        //输出数据
        writer.write(temp);
    }
    private static Set<String> defaultFieldList;
}

```

### 8.3.5 分布式搜索

当一个索引的大小超过一个机器的处理能力的时候，就需要用到分布式索引了。Solr 直接用 HTTP 来实现分布式搜索。

分布式搜索要达到如下目标：客户端应用，例如 SolrJ 对分布式搜索是完全不可知的。分布式搜索的处理和结果合并都在请求 handler 内部处理了。在结果合并以后，保持响应返回的结果格式不变。

分布式搜索的具体实现方法是：一个新的叫做 MultiSearchRequestHandler 的 Request Handler 执行对多个子搜索的分布式搜索（子搜索服务叫做 shard）。handleRequestBody 方法被分成查询构建和执行两个方法。为了增加分布式搜索功能，所有的搜索请求 handler 都扩展这个 MultiSearchRequestHandler。标准的 StandardRequestHandler 和 DisMaxRequestHandler 改成扩展这个类。

如果 shards 出现在请求参数中，分布式搜索就开始起作用了，否则搜索的是本地索引库。例如，shards=local,host1:port1,host2:port2 会搜索本地索引和两个远程索引。从三个 shard 返回的结果合并后返回给客户端。

可以通过下面的 URL 地址访问多个 Solr 实例：

```

http://localhost/select/?q=%E5%8C%BB%E7%94%9F&version=2.2&start=1000
&rows=20&indent=on&shards=local,localhost:8080

```

分布式搜索系统整体结构如图 8-8 所示。其中，需要考虑如何在多个 shard 中分布文档，也就是分布文档算法。一个简单的分布文档方法是根据文档编号分布到不同的 shard:hash(id)% numShards。如果重建索引很容易或者 shard 数量固定不变，这样是可以的。在 shard 数量变化的情况下，可考虑采用一致性散列（Consistent Hashing）。设想删除或者增加一个 shard，不会造成文档到 shard 映射的剧烈改变，这就是一致性散列要达到的效果。一致性散列的结果不只是一个数，而是一个完整的探查序列的情形。想象有一个年级的同学分在

几个班，把其中一个班解散，这个班的学生分到其他班，而其他班的学生仍旧在原来的班。

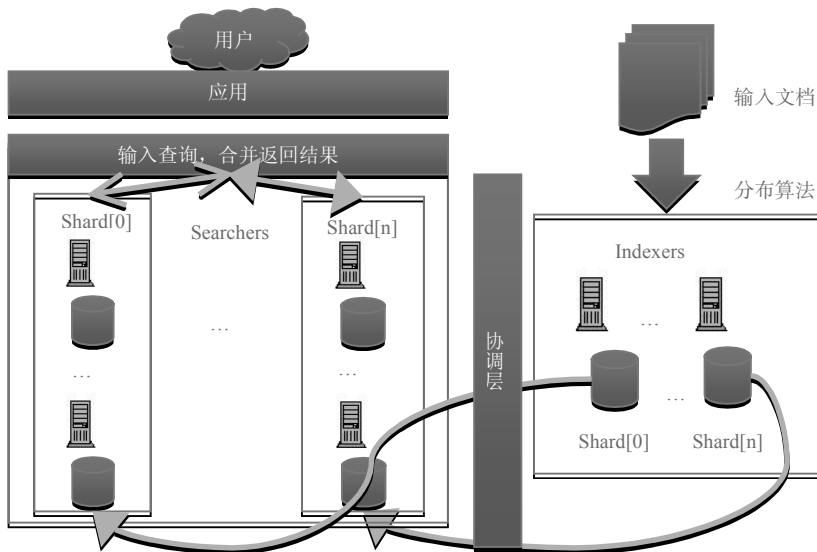


图 8-8 分布式搜索系统整体结构图

索引分布到多台机器后，提高了单点失败的可能性，也就是一台搜索服务器无法访问后，导致整个搜索结果都不可用了。可以利用虚拟 IP 地址（VIP）来避免单点失败。VIP 是一个不连接到一个指定的计算机或者计算机网卡的 IP 地址。送到 VIP 地址的输入包被重定向到物理网络接口。分布式搜索通过 HTTP 网络协议实现，因此可以查询 shard 的 VIP。通过 VIP 来实现负载均衡和容错的结构如图 8-9 所示。

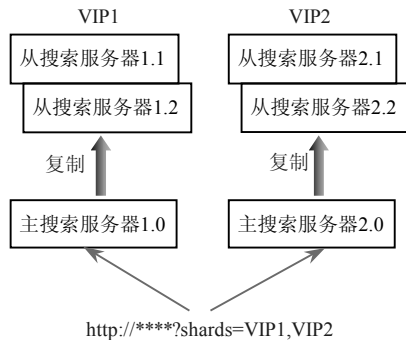


图 8-9 分布式搜索与索引复制结构图

### 8.3.6 SolrJ 查询分析器

为了支持象 AND（与）、OR（或）、NOT（非）这样的高级查询语法。Lucene 使用 JavaCC 生成的 QueryParser 类实现用户查询串的解析。SolrJ 自身没有这样的实现，下面实现一个和 Lucene 查询语法兼容的查询语法解析器。

查询分析一般用两步实现，词法分析和语法分析。词法分析阶段根据用户输入返回单词符号序列，而语法分析阶段则根据单词符号序列返回需要的查询串。

词法分析的功能是从左到右扫描用户输入查询串，从而识别出标识符、保留字、整数、

浮点数、算符、界符等单词符号，把识别结果返回到语法分析器，以供语法分析器使用。这一部分的输入是用户查询串，输出是单词符号串的识别结果。例如，对如下的输入片断：

```
title:car site:http://www.sina.com
```

词法分析的输出结果可能是：

```
TREM title
COLON :
TREM car
TREM site
COLON :
TREM http://www.sina.com
```

词法分析可以采用 JFlex 这样的工具生成，也可以手工编写。因为查询词法比较简单，所以这里采用手工实现一个词法分析。语法分析采用 YACC 的 Java 版本 BYACC/J (<http://byaccj.sourceforge.net/>)。BYACC/J 根据 YACC 源文件生成 Java 源代码。YACC 推导的返回类型 Query 定义如下所示：

```
public interface Query {
    public String getQueryType(); //取得查询类型,对应 Solr 的 Request Handler
    public String getQuery(); //取得查询串
}
```

语义值存储在一个叫做 ParserVal 的类中，因此修改 ParserVal 中属性 obj 的类型定义。

```
public class ParserVal{
    ...
    /**
     * 联合体对象的值
     */
    public Query obj;
    ...
}
```

定义 Token 的类型有如下几种：

```
%token AND OR NOT PLUS MINUS LPAREN RPAREN COLON TREM SKIP RANGEIN_START
RANGEIN_TO RANGEIN_END
```

下面是词法分析器的具体实现代码：

```
public class Yylex {
```



```
private Parser yyparser;    //解析器
private String buffer;      //查询串缓存
private int tokPos = 0;     //Token 的当前位置
private int tokLen = 0;     //Token 的长度

/**
 * 词法分析器的构造函数
 *
 * @param r 输入查询串
 * @param yyparser 解析器对象
 */
public Yylex(String r, Parser yyparser) {
    buffer = r;
    this.yyparser = yyparser;
}

/**
 * 遍历扫描直到匹配正则表达式
 * 如果结束，返回 0
 *
 * @返回下一个 token
 */
public int yylex() {
    tokPos += tokLen;
    if (tokPos >= buffer.length()) {
        //输入串解析结束
        return 0;
    }

    char ch = buffer.charAt(tokPos);
    switch (ch) {
        case '+':
            tokLen = 1;
            return Parser.PLUS;
        case '-':
            tokLen = 1;
            return Parser.MINUS;
        case '(':
        case '[':
            tokLen = 1;
            return Parser.LPAREN;
        case ')':
        case ']':
            tokLen = 1;
            return Parser.RPAREN;
```

```

case '[':
    tokLen = 1;
    return Parser.RANGEIN_START;
case ']':
    tokLen = 1;
    return Parser.RANGEIN_END;
case ':':
case ': ':
    tokLen = 1;
    return Parser.COLON;
case '|':
    if (tokPos + 1 < buffer.length() && buffer.charAt(tokPos +
1) == '|') {
        tokLen = 2;
        return Parser.OR;
    }
    tokLen = 1;
    return Parser.TREM;

case '&':
    if (tokPos + 1 < buffer.length() && buffer.charAt(tokPos +
1) == '&') {
        tokLen = 2;
        return Parser.AND;
    }
    tokLen = 1;
    return Parser.TREM;
case ' ':
case '\\t':
case ' ':
    tokLen = 1;

    return yylex();
case '"':
    tokLen = 1;
    while (tokPos + tokLen < buffer.length()) {
        char chTerm = buffer.charAt(tokPos + tokLen);
        if (chTerm != '"') {
            tokLen++;
        } else {
            tokLen++;
            break;
        }
    }
    yyparser.yylval=new ParserVal(buffer.substring(tokPos,

```



```
tokPos+tokLen));

return Parser.TREM;
default:
    tokLen = 1;
    while (tokPos + tokLen < buffer.length()) {
        char chTerm = buffer.charAt(tokPos + tokLen);
        if (chTerm != ' ' &&
            chTerm != '\t' &&
            chTerm != ' ' &&
            chTerm != '(' &&
            chTerm != ')' &&
            chTerm != ' ' &&
            chTerm != ' (' &&
            chTerm != ':' &&
            chTerm != ']') {
            if (chTerm == ':') {
                if ("http".equals(buffer.substring(tokPos,
                    tokPos+tokLen))) {
                    tokLen++;
                } else {
                    break;
                }
            } else {
                tokLen++;
            }
        } else {
            break;
        }
    }
    String cur = buffer.substring(tokPos, tokPos+tokLen);
    if (cur.equals("AND")) {
        return Parser.AND;
    }
    if (cur.equals("OR")) {
        return Parser.OR;
    }
    if (cur.equals("TO")) {
        return Parser.RANGEIN_TO;
    }
    yyparser.yylval = new ParserVal(cur);
    return Parser.TREM;
}
}
}
```

可以测试一下这个词法分析程序，看是否能返回正确的 Token 类型：

```
public static void main(String[] args) {
    String i = "title:car link:http://www.sina.com"; //输入字符串
    Parser yyparser = new Parser();
    Yylex lexer = new Yylex(i, yyparser);

    int type = 1;
    while (type!=0){
        type = lexer.yylex(); //得到 Token 类型
        String result = String.valueOf(type);
        if (yyparser!=null && yyparser.yylval!=null ){
            result+= " " + yyparser.yylval.sval; //得到 Token 值
        }
        System.out.println(result);
    }
}
```

YACC 源文件由以下三部分组成：

- DECLARATIONS 区域，在这里定义 token 和过程等；
- ACTIONS 区域，在这里定义文法和行为；
- CODE 区域，在这里定义用户方法。

三部分由一个%%行分隔开，如下所示：

```
DECLARATIONS
%%
ACTIONS
%%
CODE
```

生成 Solr 查询串的 YACC 语法文件 queryparser.y 的内容如下所示：

```
%start querystr
%token AND OR NOT PLUS MINUS LPAREN RPAREN COLON TREM SKIP RANGEIN_START
RANGEIN_TO RANGEIN_END
%left OR
%left AND
%left NOT
%left PLUS
%left MINUS
%%
```





```
querystr    : query { $$ = $1; }
;

query      : /*empty */
           | query clause {
             if ($1 == null)
             {
                 $$ = $2;
             }
             else
             {
                 BooleanQuery bq = new BooleanQuery();

                 bq.Add($1.obj, BooleanClause.Occur.SHOULD);
                 if ($2.obj instanceof BooleanQuery)
                 {
                     BooleanQuery clause = (BooleanQuery)$2.obj;
                     if (clause.plus)
                     {
                         bq.Add($2.obj, BooleanClause.Occur.MUST);
                     }
                     else if (clause.minus )
                     {
                         bq.Add($2.obj, BooleanClause.Occur.
                             MUST_NOT);
                     }
                     else
                     {
                         bq.Add($2.obj, BooleanClause.Occur.SHOULD);
                     }
                 }
                 else
                 {
                     bq.Add($2.obj, BooleanClause.Occur.SHOULD);
                 }
                 $$ = new ParserVal(bq);
             }
           }
;

clause     : clause OR clause {
           BooleanQuery bq = new BooleanQuery();
           bq.Add($1.obj, BooleanClause.Occur.SHOULD);
           bq.Add($3.obj, BooleanClause.Occur.SHOULD);
           $$ = new ParserVal( bq );
           }
```

```

        | clause AND clause {
            BooleanQuery bq = new BooleanQuery();
            bq.Add($1.obj, BooleanClause.Occur.MUST);
            bq.Add($3.obj, BooleanClause.Occur.MUST);
            $$ = new ParserVal( bq );
        }

        | LPAREN clause RPAREN {
            $$ = new ParserVal( $2.obj );
        }

        | PLUS clause {
            BooleanQuery bq = new BooleanQuery();
            bq.Add($2.obj, BooleanClause.Occur.SHOULD);
            bq.plus = true;
            $$ = new ParserVal( bq );
        }

        | MINUS clause {
            BooleanQuery bq = new BooleanQuery();
            bq.Add($2.obj, BooleanClause.Occur.SHOULD);
            bq.minus = true;
            $$ = new ParserVal( bq );
        }

        | NOT clause {
            BooleanQuery bq = new BooleanQuery();
            bq.Add($2.obj, BooleanClause.Occur.SHOULD);
            bq.minus = true;
            $$ = new ParserVal( bq );
        }

        | TREM {
            String termStr = $1.sval;
            BooleanQuery bq = new BooleanQuery();
            for (int i = 0; i < defaultSearchFields.length; ++i)
            {
                bq.Add(new TermQuery(defaultSearchFields[i],
                    termStr), BooleanClause.Occur.SHOULD);
            }
            $$ = new ParserVal(bq);
        }

        | TREM COLON TREM {
            String fieldStr = $1.sval;
            String termStr = $3.sval;
            $$ = new ParserVal( new TermQuery(fieldStr,
                termStr) );
        }

        | TREM COLON RANGEIN_START TREM RANGEIN_TO TREM RANGEIN_END{
            String fieldStr = $1.sval;

```

```

        String fromStr = $4.sval;
        String toStr = $6.sval;
        $$ = new ParserVal( new RangeQuery(fieldStr,
        fromStr,toStr) );
    }
;
%%
private Ylex lexer;
public String[] defaultSearchFields = {"title","body"}; //默认搜索列
private int yylex () {
    int yyl_return = lexer.yylex();
    return yyl_return;
}
public void yyerror (String error) {
    System.err.println ("Error: " + error);
}
public Parser(String i) {
    lexer = new Ylex(i, this);
}
public static void main(String args[]) { //测试方法
    String input = "title:中国 OR 北京";
    Parser parser = new Parser(input);
    parser.yyparse();
    System.out.println(parser.val_peek(0).obj.ToString());
}

```

用 BYACC 生成 Java 源代码：

```
$yacc -J queryparser.y
```

### 8.3.7 扩展 SolrJ

SolrJ 通过请求类和响应类提供了方便的接口用来扩展。例如实现从 URL 地址 <http://localhost/admin/stats.jsp#core> 读取索引状态，如图 8-10 所示。

CORE	
<b>name:</b>	Searcher@8d1a06 main
<b>class:</b>	org.apache.solr.search.SolrIndexSearcher
<b>version:</b>	1.0
<b>description:</b>	index searcher
<b>stats:</b>	caching : true numDocs : 4679916 maxDoc : 5937247 readerImpl : MultiReader readerDir : org.apache.lucene.store.FSDirectory@/usr/local/tomcat55/bin/gongkong/data/index indexVersion : 1188978548783 openedAt : Sun Nov 30 13:31:13 CST 2008 registeredAt : Sun Nov 30 13:31:20 CST 2008

图 8-10 索引状态

首先定义请求类:

```
public CoreRequest() {
    super( METHOD.GET, "/admin/stats.jsp#core" );
}
```

然后定义响应类:

```
public CoreResponse(NamedList<Object> res) {
    super(res);
    indexInfo = res;
}
```

返回 XML 格式内容的解析类 CoreResponseParser:

```
protected NamedList<Object> readNamedList( XMLStreamReader parser ) {
    NamedList<Object> nl = new NamedList<Object>();
    int status;
    while( parser.hasNext() ) {
        status = parser.next();
        if (status==XMLStreamConstants.START_ELEMENT) {
            String n = parser.getLocalName();
            if("stat".equals(n)) {
                //取得属性名
                n = parser.getAttributeValue(0);
                parser.next();
                //取得属性值
                String v = parser.getText().trim();
                nl.add(n, v);
            }
        }
    }
    return nl;
}
```

### 8.3.8 扩展 Solr

Solr 本身是 REST 风格的, 它通过 Servlet 响应用户请求。以 MoreLikeThis handler 为例, 在 solrconfig.xml 中包含以下内容:

```
<requestHandler name="/mlt" class="solr.MoreLikeThisHandler">
  <lst name="defaults">
    <str name="mlt.fl">manu,cat</str>
    <int name="mlt.mindf">1</int>
```



```
</lst>  
</requestHandler>
```

MoreLikeThisHandler 对应的请求方法：

```
http://localhost:8983/solr/mlt?q=id:UTF8TEST&mlt.fl=manu,cat&mlt.mindf=1&mlt.mintf=1
```

编写 Solr 的 handler 的方法需要经过以下几个步骤的操作。

- ① 将 Solr 源文件解压，在 Eclipse 中新建工程，将源文件以及相关的 jar 文件导入。
- ② 在 org.apache.solr.handler 包下（一般在此包下进行扩展）新建 Java 类。
- ③ 新建的类需要继承 RequestHandlerBase 类，并且实现其中的 handleRequestBody (SolrQueryRequest req, SolrQueryResponse rsp) 方法，同时需要覆盖 getVersion()、getDescription()、getSourceId()、getSource()、getDocs()等方法。
- ④ 在 handleRequestBody(SolrQueryRequest req, SolrQueryResponse rsp)方法中，req 表示传入的参数对象，rsp 表示经过处理后得到的需要显示的对象。
- ⑤ 根据需要编写业务处理逻辑。
- ⑥ 在 Solr 中，常量一般在 org.apache.solr.common.params 包下的接口 CommonParams 中定义。在本项目中，需要在 CommonParams 中添加新的常量，如：

```
public static final String URI = "uri";//URI 的值表示访问的参数
```

- ⑦ 以上工作实现后，对 Solr 重新打包，然后加入到 Web 项目目录\WEB-INF\lib 中。
- ⑧ 打开 Solr 文件夹（对应 resin 文件夹下的 solr 文件夹），打开 conf 文件夹下的 solrconfig.xml 文件，在<config>……</config>标签中，添加如下内容：

```
<requestHandler name="/urlinfo" class="solr.URInfoWordsHandler">  
</requestHandler>
```

其中 name 属性决定了访问的路径，class 属性决定了处理类。

- ⑨ 打开 IE 浏览器，输入“http://localhost:8081/index/urlinfo?url=? wt=?”，其中“localhost:8081/index/”表示该 WEB 应用程序的访问地址，“urlinfo”对应⑧中的

<requestHandler>标签中的 name 属性的值。url 对应传入的查询网页地址（对应 ⑥ 中添加的常量的值，即 uri）。wt 表示要输出的格式，值一般为 xml 和 json，表示输出的格式为 XML 格式或者 JSON 格式。

以下通过一个例子具体说明。

① 在合适的包下编写所需要的 Java 类，本例中类名为 URIToWordsHandler，该类继承 RequestHandlerBase 类。

```
public class URIToWordsHandler extends RequestHandlerBase {
    public void handleRequestBody(SolrQueryRequest req, SolrQuery
    Response rsp)
        throws Exception {
        /**
         * 获取传递的参数集合
         */
        SolrParams params = req.getParams();

        /**
         * 从参数集中得到想要的参数
         */
        String uri = params.get(CommonParams.URI);

        /**
         * 进行业务处理
         */
        ArrayList<String> s = new ArrayList<String>();
        s.add("大家新年好");
        s.add("今日新闻：火车票大战");
        s.add("生活");
        Words words = new Words(s, "娱乐");

        String type = words.getType();
        NamedList<Object> word = new SimpleOrderedMap<Object>();
        for (String o : words.getWord()) {
            word.add("word", o);
        }

        /**
         * 将业务处理结果添加到 rsp，并且输出
         */
        rsp.add("type", type);
        rsp.add("words", word);
    }
}
```



```

    }

    class Words {
        private List<String> word;
        private String type;

        public Words(List<String> word, String type) {
            super();
            this.word = word;
            this.type = type;
        }

        public List<String> getWord() {
            return word;
        }

        public void setWord(List<String> word) {
            this.word = word;
        }

        public String getType() {
            return type;
        }

        public void setType(String type) {
            this.type = type;
        }
    }

    ////////////////////////////////////////////////// SolrInfoMBeans methods ////////////////////////////////////////
    //////////////////////////////////////////////////以下覆写父类的方法 ////////////////////////////////////////
}

```

② 在 solrconfig.xml 中<config>标签下添加相应的配置信息：

```

<requestHandler name="/urlinfo" class="solr.URInfoWordsHandler">
    <!-- /urlinfo 表示访问路径 class 表示对应的处理类 -->
    <!-- 以下为其他说明，可根据需要修改或删除，此信息将会在网页显示 -->
    <lst name="其他说明">
        <int name="名称">参数</int>
    </lst>
</requestHandler>

```

③ 启动服务器，在浏览器地址栏输入 “http://localhost:8081/index/urlinfo?url=

d&wt=xml”。其中，urlinfo 对应 b 项中提到的访问路径，url 表示需要传递的参数，wt 表示返回的结果格式。如图 8-11 所示为 wt=xml 时的显示结果。

```
<?xml version="1.0"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">16</int>
  </lst>
  <str name="type">娱乐</str>
  <lst name="words">
    <str name="word">大家新年好</str>
    <str name="word">今日新闻：火车票大战</str>
    <str name="word">生活</str>
  </lst>
</response>
```

图 8-11 wt=xml 时的显示结果

当在浏览器地址栏输入 URL 地址 “http://localhost:8081/index/urlinfo?url=d&wt=json” 时，指定返回结果格式是 JSON，显示结果如图 8-12 所示。

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 0,
    "type": "娱乐",
    "words": {
      "word": "大家新年好",
      "word": "今日新闻：火车票大战",
      "word": "生活"
    }
  }
}
```

图 8-12 wt=json 时的显示结果

### 8.3.9 查询 Web 图

为了支持在 Solr 中以 “link:” 语法查找链接网页，写一个 LinkToThisHandler 的 Solr 插件调用第 2 章已经实现的 WebGraph。

```
public final static String PREFIX = "lt.";
public final static String URL_FIELD = PREFIX + "fl";
private static final String INDEX_DIR = PREFIX + "dbDir";

private WebGraph webGraph = null;

@Override
public void init(NamedList args) {
    super.init(args);
    SolrParams p = SolrParams.toSolrParams(args);
    String dir = p.get(INDEX_DIR);
    try {
        //读入 WebGraph
        webGraph = new WebGraph(dir);
    } catch (DatabaseException e) {
        throw new RuntimeException("Cannot open WebGraph database", e);
    }
}

@Override
public void handleRequestBody(SolrQueryRequest req, SolrQueryResponse
    rsp) throws Exception {
    SolrParams params = req.getParams();
    SolrIndexSearcher searcher = req.getSearcher();
    String q = params.get(CommonParams.Q);

    SolrParams required = params.required();
    //取得 URL 列的名称
    String field = required.get(LinkToThisHandler.URL_FIELD);
```





```
//需要返回的字段
String fl = params.get(CommonParams.FL);
int flags = 0;
if (fl != null) {
    flags |= SolrPluginUtils.setReturnFields(fl, rsp);
}

int start = params.getInt( CommonParams.START, 0 );
int rows = params.getInt( CommonParams.ROWS, 10 );

DocList docList = null;

if( q != null ) {

    //找到一个基础的匹配
    Query query = QueryParsing.parseQuery(q, params.get(CommonParams.DF),
        params, req.getSchema());
    DocList match = searcher.getDocList(query, null, null, 0, 1, flags);

    //DocIterator 是一个迭代器，但这里只处理第一个匹配
    DocIterator iterator = match.iterator();
    if( iterator.hasNext() ) {
        //在结果中的每一个文档里都做这样一个请求处理
        int id = iterator.nextDoc();
        IndexReader reader = searcher.getReader();
        Document doc = reader.document(id);
        String toURL = doc.getField(field).stringValue();
        String[] fromURLs = webGraph.inLinks(toURL);
        ArrayList<Integer> docIds = new ArrayList<Integer>(fromURLs.length);

        for(int i = 0;i<fromURLs.length;++i)    {
            TermQuery tq = new TermQuery(new Term(field,fromURLs[i]));
            Hits hits = searcher.search(tq);
            if(hits.length()>=1)    {
                int docId = hits.id(0);
                docIds.add(docId);
            }
        }
        int[] ids = new int[docIds.size()];
        for(int i=0;i<ids.length;++i)    {
            ids[i]=docIds.get(i);
        }
        DocSlice result = new DocSlice(0,ids.length,ids,null,ids.
            length,0);
        docList = result.subset(start, rows);
    }
}
else {
```

```

        throw new SolrException( SolrException.ErrorCode.BAD_REQUEST,
            "MoreLikeThis requires either a query (?q=) or text to find
            similar documents." );
    }
    if( docList == null ) {
        docList = new DocSlice(0,0,null,null,0,0);
    }
    rsp.add( "response", docList );
}

```

在 SolrConfig.xml 中配置 LinkToThisHandler:

```

<requestHandler name="/lt" class="solr.LinkToThisHandler">
  <lst name="defaults">
    <str name="lt.fl">url</str>
  </lst>

  <!-- Main init params for handler -->

  <!--下面写入 webgraph 数据库所在的路径-- -->
  <str name="lt.dbDir">webgraph</str>

</requestHandler>

```

然后我们就可以在浏览器中测试效果了:

```

http://localhost/solr/lt/?q=url%3Ahttp%5C%3A%5C%2F%5C%2Fwww.baotron.
com%5C%2Findex.asp&version=2.2&start=100&rows=10&indent=on

```

Solr 内部调用 LinkToThisHandler, 查询哪些网页指向了 “http://www.baotron.com/Findex.asp”。



## 8.4 本章小结

本章介绍了企业级的搜索服务器 Solr, 从它的基本用法到对 Solr 服务器和 Solr 客户端的扩展。通过仔细的调整性能, 可以使用 Solr 实现搜索若干个 TB 的文档内容。

和 Lucene 相比, Solr 有更多的缓存支持, 并且支持主从复制和分片两种分布式部署方式。2009 年年底发布的 Solr 1.4 集成了在搜索结果中聚类功能。聚类功能本身使用另外一个开源项目 Carrot2 (<http://project.carrot2.org/>) 完成。

Solr 通过 ZooKeeper 自动管理大的搜索服务器集群。



## 第 9 章

# 地理信息系统案例分析

电子地图是车载导航和互联网上的重要应用。地图上有些用户关注的兴趣点，例如一个餐馆就是一个兴趣点。兴趣点也叫做 POI（Point of Interest），每个 POI 包含名称、地址、类别、经度、纬度等信息。

数据采集是地图数据更新和专业的地理位置信息获取的重要方法，需要把现实中的专业地理位置信息准确地标注在地图上。采集用于地图的地理位置信息，需要保证地图数据的准确性和更新的及时性。借助人工作采集数据来取得 POI 信息，获取数据成本高，所以需要考局局部更新的问题。

从新闻中发现更新信息的流程是：首先抓取新闻，然后提取其中的标题和正文等信息，再从中提取 POI 信息，最后排重后输出。



### 9.1 新闻提取

需要从任意新闻网页中提取标题和正文等信息。这里采用信息提取的方法来实现。首先定义要提取的数据结构：

```
public class NewsInfo {  
    String title;           //标题
```

```

String text;           //正文
String author;         //作者
String infoSource;     //信息来源
String publishDate;    //发布日期
}

```

定义标注网页中的文本的类型：

```

public enum DocType {
    Start,               //开始状态
    End,                 //结束状态
    PrefixAuthor,        //作者前缀
    SuffixAuthor,        //作者后缀
    PrefixTitle,         //标题前缀
    SuffixTitle,         //标题后缀
    Title,               //标题
    PrefixText,          //正文前缀
    SuffixText,          //正文结尾
    Text,                //正文
    PrefixInfoSource,    //消息来源前缀
    InfoSource,          //消息来源
    PrefixPublishDate,   //发布时间前缀
    PublishDate,         //发布时间
    Time,                //表示时分秒的时间
    Date,                //表示年月日的时间
    Author,              //作者
    Unknow,              //未知
    Other,               // 其他
}

```

去掉网页标题中开始和结尾的噪音。例如：网页中的 HTML 原始标题是“日本地震已造成 33 人死亡\_滚动新闻\_新浪财经\_新浪网”，其中结尾部分是噪音。提取标题的实现如下：

```

//取得净化后的标题
public String getClearTitle(Node node) {
    int startPos = 0;
    int endPos = 0;
    String title = "";
    StringBuffer sb = new StringBuffer();
    getTitle(sb, node); //取得网页中的原始标题

    ArrayList<DocToken> ret = DocTagger.tag(sb.toString());

    boolean findTitleStartFlag = false;
}

```



```
boolean findTitleEndFlag = false;
for (int i = 0; i < ret.size(); ++i) { //根据标注类型找有效的标题信息
    if (ret.get(i).type.equals(DocType.PrefixTitle)
        && (findTitleStartFlag == false)
        && (findTitleEndFlag == false)) {
        findTitleStartFlag = true; /* 找到标题开头标志 */
        startPos = i;
        continue;
    } else if (((ret.get(i).type.equals(DocType.SuffixTitle))
        || (ret.get(i).type.equals(DocType.PublishDate)))
        && (findTitleEndFlag == false)) {
        findTitleEndFlag = true; /* 找到标题结尾标志 */
        endPos = i;
        continue;
    }
}

/* 根据发现标题的标志进行处理 */
if (findTitleStartFlag == true) {
    if (findTitleEndFlag == true) {
        /* 从 title 的 start 标志 拷贝至 end 标志 */
        for (int j = startPos + 1; j < endPos; j++) {
            if (!ret.get(j).type.equals(DocType.Start)
                && !ret.get(j).type.equals(DocType.End)
                && !ret.get(j).type.equals(DocType.Link)
                && !ret.get(j).type.equals(DocType.PrefixTitle)
                && !ret.get(j).type.equals(DocType.SuffixTitle)) {
                title += ret.get(j).termText;
            }
        }
    } else {
        /* 从 title 的 start 标志 拷贝至 结尾 */
        for (int j = startPos + 1; j < ret.size(); j++) {
            if (!ret.get(j).type.equals(DocType.Start)
                && !ret.get(j).type.equals(DocType.End)
                && !ret.get(j).type.equals(DocType.Link)
                && !ret.get(j).type.equals(DocType.PrefixTitle)
                && !ret.get(j).type.equals(DocType.SuffixTitle)) {
                title += ret.get(j).termText;
            }
        }
    }
} else {
    if (findTitleEndFlag == true) {
        for (int j = 0; j < endPos; j++) {
```

```

        if (!ret.get(j).type.equals(DocType.Start)
            && !ret.get(j).type.equals(DocType.End)
            && !ret.get(j).type.equals(DocType.Link)
            && !ret.get(j).type.equals(DocType.PrefixTitle)
            && !ret.get(j).type.equals(DocType.SuffixTitle)){
            title += ret.get(j).termText;
        }
    }
} else {
    title = sb.toString().trim();
}
}

return title;
}

```

去掉网页正文中开始和结尾的噪音。提取正文的代码如下所示：

```

static NewsInfo extractAll(Node documentNode) {
    NewsInfo needInfo = new NewsInfo();
    String infoSource = ""; /* 信息来源 */
    String publishDate = ""; /* 发布时间 */
    String title = ""; /* 标题 */
    String text = ""; /* 正文 */
    String author = ""; /* 作者 */
    int start = 0; //正文的开始位置
    int end = 0; //正文的结束位置

    boolean findTextStartFlag = false; /* 初始化找到正文开头标志 */
    boolean findTextEndFlag = false; /* 初始化找到正文结尾标志 */
    boolean findInfoSource = false; /* 初始化找到信息来源标志 */
    boolean findPublishDate = false; /* 初始化找到发布日期标志 */
    boolean findTextAuthor = false; /* 初始化找到作者标志 */

    /* 获取标题 */
    title = getClearTitle(documentNode);

    /* 取得 DOM 对象内的所有文本 */
    String content = textExtractor(documentNode);

    /* 执行信息提取 采用规则方式提取小段文本*/
    ArrayList<DocToken> ret = DocTagger.tag(content);

    for (int j = 0; j < ret.size(); j++) {
        DocToken token = ret.get(j);
    }
}

```



```
/* 信息来源 */
if (token.type.equals(DocType.InfoSource)
    && (findInfoSource == false)) {
    infoSource = token.termText;
    findInfoSource = true; /* 找到信息来源标志 */
    continue;
}

/* 作者 */
if (token.type.equals(DocType.Author) && (findTextAuthor ==
false)) {
    author = token.termText;
    findTextAuthor = true; /* 找到作者 */
    if (author.length() >= 4) {
        author = "";
        findTextAuthor = false;
    }
    continue;
}

/* 发布时间 */
if (token.type.equals(DocType.PublishDate)
    && (findPublishDate == false)) {
    publishDate = token.termText;
    findPublishDate = true; /* 找到发布日期标志 */
    continue;
}

/* 正文前缀 */
if (token.type.equals(DocType.PrefixText)
    && (findTextStartFlag == false)) {
    start = j;
    findTextStartFlag = true;
    continue;
}

/* 正文后缀 */
if (token.type.equals(DocType.SuffixText)
    && (findTextEndFlag == false)
    && (findTextStartFlag == true)) {
    end = j;
    findTextEndFlag = true;
    continue;
}
```

```

    }

    text = "";
    if ((findTextStartFlag == true) && (findTextEndFlag == true)) {
        for (int j = start + 1; j < end - 1; j++) {
            text += ret.get(j).termText;
        }
    }

    // 去掉头尾的一些不需要的字符
    title = Entities.HTML40.unescape(title).trim();

    /* 处理其中的转义字符 */
    text = Entities.HTML40.unescape(text.toString()).trim();
    needInfo.title = title;
    needInfo.text = text;
    needInfo.author = author;
    needInfo.infoSource = infoSource;
    needInfo.publishDate = publishDate;

    return needInfo;
}

```



## 9.2 POI 信息提取

在城市发展建设过程中,地图数据信息的变更不可避免,这就需要及时更新数据。监测新闻等网络媒体中涉及地址变更的信息。通过爬虫抓取最新的和地图数据信息相关的变更信息。通过信息提取技术形成变化的地址信息列表。需要提取的 POI 相关信息有:

- POI 主体:例如深圳发展银行永康支行。
- 所属地区:例如杭州、深圳福田区等。需要用行政区域编码来确定区域。
- 时间:例如 2009 年 04 月 28 日等时间性的词。
- 变更事件:如开业、拆迁、完工、迁址、关门等动词。

从新闻中提取描述主体与事件等结构化信息。例如下面这篇新闻:

刘老根大舞台北京开业 郭德纲笑言无竞争会捧场 2009 年 04 月 28 日 17:49 来源:武汉晚报 郭德纲的德云社就在“刘老根大舞台”前门剧场的附近,两者同为逗乐艺术,竞争和比较在所难免。赵本山认为,二人转和相声有不同的欣赏观众,大家可以在竞争中求发展,在发展中成为好朋友,相互借鉴和促进。



提取出如下特征。

- 区域：北京
- 主体：刘老根大舞台
- 事件：开业

从上面的例子可以看到，感兴趣的主体、事件、区域全部都在标题中出现了。对于这样的可以采用整体提取的方式，把这些信息提取到一个 POIInfo 类的实例。

```
public class POIInfo {  
    public DocNode poi;    //主体  
    public DocNode place;  //区域  
    public DocNode matter; //事件  
}
```

还有些信息比较分散，这时候需要根据整篇新闻进行全局考虑，提取出全局性的主体、区域和事件。

为了降低实现的复杂度，基于分而治之的思想，把信息提取分成 5 个阶段的管线：

- 中文分词；
- 词性标注；
- 名字发现；
- 句法分析，一般限于名词和动词短语识别，一个句法树的例子如图 9-1 所示；
- 语义解释，一般基于模式匹配。

然而，管线结构容易导致错误累积。例如，一个标注中的错误可能导致句法分析失败，接着导致语义解释失败，这类似于多米诺骨牌效应。

集成的模型可以限制错误传播，通过联合起来做所有的决策。因此，应设计一个集成的模型，在这个模型里，标注、名字发现、解析和语义解释决策都有机会相互影响。

输入一篇新闻，返回的是一个 DocNode 组成的数组。

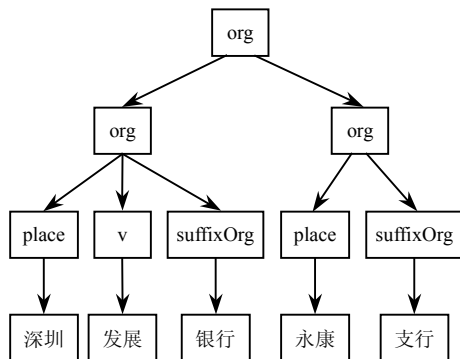


图 9-1 主体句法树

```

public class DocNode {
    public String termText;        //词
    public DocType type;          //类型
    public int start;              //开始位置
    public int end;                //结束位置
    public long cost;              //概率
    public ArrayList<DocNode> children = new ArrayList<DocNode>();
                                    //孩子节点
}

```

在匹配模式的过程中生成树的代码如下所示:

```

public static void replace(ArrayList<DocNode> key, int offset,
                           ArrayList<DocSpan> spans) {
    int j = 0;
    for (int i = offset; i < key.size(); ++i) {
        DocSpan span = spans.get(j);
        DocNode token = key.get(i);
        StringBuilder newText = new StringBuilder();
        int newStart = token.start;
        int newEnd = token.end;
        DocType newType = span.type;

        /* 组合新生成的节点 */
        for (int k = 0; k < span.length; ++k) {
            token = key.get(i + k);
            newText.append(token.termText);
            newEnd = token.end;
        }
        /* 准备新生成的结点并将老结点挂接在新节点的孩子结点上 */
        DocNode newToken = new DocNode(newStart, newEnd,
                                         newText.toString(), newType);

        /* 将老节点从队列中删除,并临时存储起来,作为将要替代的新节点的孩子结点 */
        for (int k = 0; k < span.length; ++k) {
            /* 如果长度大于1 或者类型不相同同时增加孩子结点 */
            if ((span.length > 1) || (!span.type.equals(token.type))) {
                newToken.children.add(key.get(i));
            }
            key.remove(i);
        }
        key.add(i, newToken);
        j++;
        if (j >= spans.size()) {

```



```
        return;  
    }  
}  
}
```

此外，依存句法可以直接发现词之间的关系，所以对于信息提取很有用。例如“上海华普大连金龙 4S 店即将于 12 月 19 日隆重开业”，这里的“即将于 12 月 19 日隆重”都是用来修饰“开业”的。依存语法认为词之间的关系是有方向的，通常是一个词支配另一个词，这种支配与被支配的关系就称作依存关系。

24 个依存关系定义如下：

```
public enum DependencyRelation {  
    ATT,    //定中关系(attribute)  
    QUN,    //数量关系(quantity)  
    ROOT,   //核心  
    COO,    //并列关系(coordinate)  
    APP,    //同位关系(appositive)  
    LAD,    //前附加关系(left adjunct)  
    RAD,    //后附加关系(right adjunct)  
    VOB,    //动宾关系(verb-object)  
    POB,    //介宾关系(preposition-object)  
    SBV,    //主谓关系(subject-verb)  
    SIM,    //比拟关系(similarity)  
    VV,     //连动结构(verb-verb)  
    CNJ,    //关联结构(conjunctive)  
    MT,     //语态结构(mood-tense)  
    IS,     //独立结构(independent structure)  
    ADV,    //状中结构(adverbial)  
    CMP,    //动补结构(complement)  
    DE,     //“的”字结构  
    DI,     //“地”字结构  
    DEI,    //“得”字结构  
    BA,     //“把”字结构  
    BEI,    //“被”字结构  
    IC,     //独立分句(independent clause)  
    DC;     //依存分句(dependent clause)  
}
```

依存文法由一系列规则组成。规则类定义如下所示：

```
public class Rule {  
    DocType[] dependents;    //被支配词性序列
```

```

int headId;           //支配词性位置
DependencyRelation type; //依存关系类型
public Rule(DocType[] deps,int governor,DependencyRelation dr){
    dependents = deps;
    headId = governor;
    type = dr;
}
}

```

MSTParser (<http://www.seas.upenn.edu/~stroetlrm/MSTParser>) 是一个 Java 实现的依存句法分析包。其中对训练实例的定义如下所示:

```

public class DependencyInstance {
    public String[] terms;      //词
    public DocType[] postags;   //标注类型
    public int[] heads;         //每个元素的头 ID
    public String[] deprels;    //依赖关系,例如"SUBJ"
}

```

训练数据中的每个句子用 3 到 4 行表示,一般的格式是:

w1	w2	...	wn
p1	p2	...	pn
l1	l2	...	ln
d1	d2	...	dn

- w1 ... wn 是以空格分开的句子中的单词。
- p1 ... pn 是每个单词的词性标注。
- l1 ... ln 是依赖关系类型标注。
- d1 ... dn 用整数表示每个单词的父亲的位置。

例如,句子“武汉取消了 49 个收费项目”用这种格式表示是:

武汉	取消	了	49	个	收费	项目
ns	vg	u	m	q	vn	n
SBV	ROOT	MT	QUN	ATT	ATT	VOB
2	0	2	5	8	8	2

这样可以识别出这个句子的主干是: 武汉 取消 了 项目。



### 9.2.1 提取主体

把主体看成是短语。传统的方法是首先进行分词和标注，然后再做短语识别。如果有多个可能的主体，则按照每个主体在文本中出现的次数，把出现次数多的作为主体。

但有些真正的主体出现次数较少，这时候怎么办？规则匹配出来的时候可以设置不同的权计，然后加权算主体。

```
public class DocSpan {
    public int length;           //覆盖的长度
    public DocType type;         //token 类型
    public int weight=0;         //权重

    public DocSpan(int l,DocType t){
        length = l;
        type = t;
    }

    public DocSpan(int l,DocType t,int weight){
        length = l;
        type = t;
        this.weight=weight;
    }

    public String toString(){
        return type+":"+length+":"+weight;
    }
}
```

写规则的时候可以增加识别出来的主体的权重，例如：

```
lhs.add(new DocSpan(6, DocType.SmallAddress,100));
//最后这个 100 是权重，可以调高
```

新闻经常在标题中描述发生一个事件的主体，如酒店开业中的酒店名、道路开通中的道路名。“主体”和“事件”连在一起的，可以增加提取出来的权重。

虽然主体一般是一个，但是也可以有多个。例如新闻内容是：

目前，佛山市有佛山皇冠假日（原佛山宾馆）、华夏新中源、南海名都、顺德哥顿、乐从财神酒店 5 家挂牌五星...

这里有5个主体：佛山皇冠假日、华夏新中源、南海名都、顺德哥顿、乐从财神酒店。把这些做成并列主体，也就是说并列的主体可以用标点“、”连接。

有些主体中出现的地名需要简化成地名简称，例如：“武汉石家庄高速”简称“武石高速”。一般取地名的第一个字作为简称，但有些则不是，例如北京简称“京”，天津简称“津”。Abbreviation.txt 存放简称词典，每行一个简称，样例如下：

```
北京市:京  
上海市:沪  
天津市:津  
重庆市:渝
```

“蚌淮高速”和“淮蚌高速”都是指同一个高速公路。解决方法是如果“高速”前的两个字都是简称，则按字符排序后输出成统一的格式“淮蚌高速”。高速公路中出现的地名一般是前后次序无关的。

因此，把高速路名称标准化：

```
String getStandName(DocNode roadName){  
    StringBuffer sb=new StringBuffer();  
    DocNode firstNode = roadName.children[0];  
    String abbreviation = abbreviationMap.get(firstNode.termText);  
    if(abbreviation!=null){  
        sb.append(abbreviation);  
    } else {  
        sb.append(firstNode.termText);  
    }  
    DocNode secondNode = roadName.children[1];  
    abbreviation = abbreviationMap.get(secondNode.termText);  
    if(abbreviation!=null){  
        sb.append(abbreviation);  
    } else {  
        sb.append(secondNode.termText);  
    }  
    return sb.toString();  
}
```

### 9.2.2 提取地区

地名有国外地名和国内地名。如果不需要关注国外地名，则可以把国外地名作为停用词去掉。和地址相关的类型有：直辖市(Municipality)、省(Province)、市(City)、区(County)、



镇（Town）、街（Street）。下面的代码根据搜集到的地名数组返回一个最重要的地名。

```
//输入候选地区，返回地区 and 这个地区对应的行政区域编码
public static PlaceAndCode getPlace(ArrayList<DocToken> places) {
    LinkedHashMap<String,Integer> address=new LinkedHashMap<String,
    Integer>();
    HashMap<String,Long> placeCode = new HashMap<String,Long>();

    for(DocToken place:places){
        String p = place.termText;
        Long pc1 = place.code;

        placeCode.put(p, pc1);

        //迭代出一个 place 对象 并用此对象与 places 集合中的所有对象进行比较
        for(DocToken place1:places){
            Long pc2 = place1.code;

            //比较它们的行政区域编码计算相似度
            int i = codeDistance(pc1,pc2);

            if(i!=0){
                Integer freq = address.get(p);

                if (freq!=null) {
                    address.put(p, freq+i);
                } else {
                    address.put(p, i );
                }
            }
        }
    }

    /**
     * 遍历地址，取出出现频率最高的那个地址
     */
    int max=0;
    String bestPlace = "";

    for(Entry<String, Integer> e : address.entrySet()) {
        String key=e.getKey();

        Integer val = e.getValue();
        if (val>max) {
            max=val;
        }
    }
}
```

```

        bestPlace=key;
    }
}

String place = SynonymReplace.replace(bestPlace);
Long codeValue = placeCode.get(place);
long code = 0;
if(codeValue != null){
    code = codeValue;
}

return new PlaceAndCode(place,code);
}

```

### 9.2.3 指代消解

有的描述对象只在内容中出现了一次或少数几次，在内容中更多的地方以代词出现。这时候，需要用指代消解（Coreference Resolution）来准确地处理这样的描述对象。和主体相关的例子如下：

昨天（20 日）晚上 10 时，随着最后一段接缝处合龙段混凝土浇筑完毕，由上海建工基础公司承建的新河南路桥实现了提前 10 天合龙贯通。预计到今年 8 月，这座桥的部分车道将通车，以缓解河南路的交通压力。

希望提取出“新河南路桥”，而这里，“这座桥”指代了“新河南路桥”，很多代词如果不还原出来，就很难抽取到内容。因为“新河南路桥”这个词只出现了一次，所以要通过指代消解来确定谈话主体。

指代消解对词性标注和语法分析有一定的依赖，判断是否消解不是根据这个词出现了几次，而是要先找到指示词“这座桥”。

用选出的先行词（antecedent）替换指代词，即进行指代消解。

具体实现上首先找出指示词的候选先行词，然后计算候选的先行词和指示词的一致性。例如“新河南路桥”和“这座桥”词尾都是“桥”，所以一致性比较高。指代语过滤器用于判断是否应该将一些实体描述作为其先行语。最后把指示词替换成对应的先行词。

下面举几个和地名相关的例子。

成都人民南路主车道完成施工 ... 记者从市重大办了解到，自人民南路综合改造工程





动工以来，我市在确保施工质量的前提下先后进行过多次改造进度提速：9 月上旬，各施工单位基本完成东侧道路路面改造，开始倒边向道路西侧打围，进行西侧道路的半幅施工；仅用 1 个月时间就完成了西侧道路的改造，在本月 15 日之前形成主车道全线通车能力，并完成天府广场至二环路范围内的人行道铺装，为西博会的顺利召开提供交通保障。

在这里“我市”指代“成都”。

东莞阳光网 ... 莞深高速三期石碣段即东江大桥于上月 28 日通车以来，受到市民的广泛欢迎。特别是我市东部镇街的市民经此去广州以及白云机场，将比走广深高速节省半个小时。

在这里“我市”指代“东莞”。

我市金龙大桥长田隧道正式通车 该工程于 2008 年 1 月开工，建成通车后，将有效改善达州天然气能源化工基地连通主城区的交通面貌，对推动化工产业园区建设，拓展城市发展空间，构建城市交通内联外接骨架网络具有重大重义。

在这里“我市”指代“达州”。

DocFactory 的 resolutionPlace 方法实现的值转换代码如下所示：

```
//对输入 tokens 中的代词执行指代消解
public static void resolutionPlace(ArrayList<DocToken> tokens) {
    for (int i=0;i<tokens.size();++i ) {

        DocToken pt = tokens.get(i);
        if (pt.type == DocType.PronounPlaces) {
            //如果检测到代词，则尝试消解

            boolean find = false;
            for(int k=i-1;k>=0;--k) { //先往前找
                DocToken predecessor = tokens.get(k);

                if (predecessor.type == DocType.Address) {
                    //找到前驱词
                    tokens.set(i, predecessor);
                    find = true;
                    break;
                }
            }
            if(!find){
```





### 9.3.1 词对齐

可以从对齐语料库中挖掘这样的双语词典。找中英文对照词表的方法术语上叫做词对齐（word alignment）。最简单的词对齐就是从前往后逐个词对应上。例如“北京 地铁”翻译成“Beijing Subway”，其中前后两个词是按顺序对齐的。

```
HashMap<String,String> wordMap = new HashMap<String,String>(); //词语对照表

String enSent = "Beijing Subway"; //英语句子
String cnSent = "北京 地铁"; //中文句子

StringTokenizer enTokenizer = new StringTokenizer(enSent); //用空格分割英文句子
StringTokenizer cnTokenizer = new StringTokenizer(cnSent); //用空格分割英文句子
while(enTokenizer.hasMoreElements()){ //有更多的词没遍历完
    wordMap.put(cnTokenizer.nextToken(),enTokenizer.nextToken());
}

for(Entry<String, String> e:wordMap.entrySet()){
    System.out.println(e.getKey()+":"+e.getValue());
}
```

把这种词对齐方法叫做 **BaselineWordAligner**。首先用 40 万条中英文对照的公司名记录作为训练样本。例如：“中钢贸易公司”对应“SINOSTEEL TRADING COMPANY”。

只需要把“中钢贸易公司”切分成：[中钢][贸易][公司]，然后运行 **BaselineWordAligner**。提取出中英文对照词表：

```
中钢:SINOSTEEL
贸易:TRADING
公司:COMPANY
```

首先假设任何一个中文词翻译到任何一个英文词的概率都是相等的。然后根据常用中英文对照词表知道“中国”可以翻译成“CHINA”。

从训练集中可以发现“中国原子能工业公司”翻译成“CHINA NUCLEAR ENERGY INDUSTRY CORP.”。“中国林木种子公司”翻译成“CHINA NATIONAL TREE SEED CORP.”。因为“公司”和“CORP.”共同出现，所以把“公司:CORP.”加入到对照词表。这样补充

中英文对照词表的方式叫做无监督的学习。

严格来说,可以采用互信息公式找出互信息最大的中英文词语对。例如,计算把“公司”翻译成“company”的互信息公式是:

$$I(company, 公司) = \log_2 \frac{P(company, 公司)}{P(company)P(公司)}$$

如果  $P(company) = 0.1$ ,  $P(公司) = 0.2$ ,  $P(company, 公司) = 0.1$ , 则计算信息熵的程序如下:

```
double entropy = (Math.log(0.1) - Math.log(0.1) - Math.log(0.2)) /
Math.log(2);
System.out.println(entropy); //输出 2.32
```

有一些现成的词对齐工具,例如 `berkeley alignment`(<https://code.google.com/p/berkeleyaligner/>)。<http://code.google.com/p/tdx-nlp/>包含一个简化版本的词对齐。除了采用词对齐,还可以用爬虫抓在线翻译的结果,补充数据。

### 9.3.2 翻译公司名

除了地名,还需要把公司名翻译成英文。例如,“潍坊英轩实业有限公司”翻译成:“Weifang YinXuan Industrial co., LTD”。“英轩”这样的企业品牌词往往不在词表中,需要采用识别规则识别出这样的未登录词。识别未登录词以后,可以把公司名切分成:“潍坊/City 英轩/Keyword 实业/Feature 有限公司/Function”,然后把关键词“英轩”翻译成拼音YinXuan,翻译其他的词靠查双语词典。

双语词典包括几种不同类型的词:

- 功能词表,存放在 `Function.txt`。
- 特征词表,存放在 `Feature.txt`。
- 一些地名相关的词表,例如:城市词表存放在 `city.txt`、区/县词表存放在 `county.txt`。

可以把词条放入 Trie 树词典。按中文组织成 Trie 树。可结束节点中包含了中文词对应的英文单词,如图 9-2 所示。

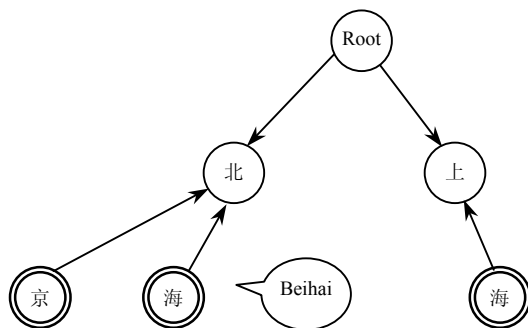


图 9-2 翻译词典 Trie 树

可结束节点中包含的词条定义如下：

```

public class CompanyEntry {
    public CompanyType type;    //类型
    public int freq;           //词频
    public String english;      //英文单词
}

```

部分词的类型如下：

```

public enum CompanyType {
    Country        //国家
    ,Municipality  //直辖市
    ,Province      //省
    ,City          //市
    ,county        //区
    ,Function      //功能词
    ,KeyWord       //关键字
    ,Feature       //行业特点
    ,Unknow       //未知
}

```

“北京市\*\*有限公司”翻译成“Beijing \*\* Ltd”，不用翻译“市”成为“city”。下面这行代码调用 `addWord` 方法把“北京市”加入词表：

```

// “北京市”翻译成“beijing”，公司类型是Municipality，权重是100
addWord("北京市", "beijing", CompanyType.Municipality, 100);

```

有些公司名的关键字不是拼音的，比如有的“三星”翻译成“samsung”。“北京三星电子”中的“三星电子”翻译成“Samsung Electronics”，而“北京三星博文科技发展有限责

任公司”中的“三星博文”翻译成“Three-star Balvin”。

提取二元连接对应的翻译结果。例如，“三星@博文”对照“Three-star Balvin”，“三星@电子”对照“Samsung Electronics”。

### 9.3.3 调整语序

对于更复杂的例子：“北京市海淀区龙岗路 35 号”翻译成“NO.35 Longgang Road Qinghe Haidian District, Beijing”。涉及到语序调整时，可以生成地名的中文依存树，然后把中文依存树转换到英文依存树。

“蒙古大使馆”不能翻译为“Mongolian Embassy”，而是“Embassy of Mongolian”，表示这个大使馆是给蒙古国大使用的。

用特征结构表示出“蒙古大使馆”。

机构名: 大使馆

地区: 蒙古

然后调用 toSentence 方法，得到“大使馆 of 地区”。

```
FeatureStructure fs = new FeatureStructure(); //特征结构
fs.organization = "Embassy";
fs.area = "Mongolian";
System.out.print(fs); //输出: Embassy of Mongolian
```

也可以用依存树来解决翻译成“Embassy of Mongolian”的问题。“蒙古”和“大使馆”之间存在“属于”依存关系。然后转换这个小的依存树，转换的时候判断依存类型，把“属于”依存类型转换成英文输出：“\*\* of \*\*”。

```
//蒙古 大使馆 的 依存文法树
//蒙古 -> 大使馆
CnDepTree area = new CnDepTree(new TreeNode("蒙古", "Mongolian", 2));
CnDepTree org = new CnDepTree(new TreeNode("大使馆", "Embassy", 3));

ArrayList<CnDepTree> struct = new ArrayList<CnDepTree>();
struct.add(area);
struct.add(org);
org.order = struct;
area.root.governor = org.root;
```



```
area.root.relation = DependencyRelation.atr;

CnDepTree cnTree = org;

EnDepTree targetTree = cnTree.convert(); //得到翻译结果
System.out.println(targetTree.toSentence());
//输出: Embassy of Mongolian
```

特征结构适合处理有省略的情况。如果没有省略，则可以使用依存树。



## 9.4 本章小结

本章介绍了从新闻中发现 POI 更新信息的方法和实现。首先对新闻分词，然后分别从标题和内容中发现 POI 相关信息，最后整理成结构化的形式输出。此外还介绍了如何把 POI 信息从中文翻译到英文。



## 第 10 章

# 户外活动搜索案例分析

自助旅行一般会结伴而行，所以驴友们在各种网站发布了自助游活动。在北京，如果要找最近一个周末爬香山的活动，直接输入这样的关键词查询通用的搜索引擎未必能够得到满意的结果。所以有人提议开发一个户外活动搜索。



### 10.1 爬虫

考虑到前期搜索用户并不多，主要是浏览网页的方式访问。所以可以由爬虫直接把一些景点介绍写入到维基。需要把维基中的图片及格式信息保留下来。

为了节省带宽，有些更新频率不高的信息在本地抓取，然后再把数据传到搜索服务器上。例如旅游攻略或线路信息等。户外活动或者旅游新闻则需要实时采集和更新。户外活动需要关注天气信息。为了快速访问天气信息，需要每天抓取天气预报。

一个强大的爬虫系统，不仅要能够及时发现每天互联网上新产生的网页，还要能够及时地更新已经抓取的网页，使之最大程度上与互联网上真实存在的页面一致。

对于这个旅游搜索系统，获取网页信息的方式有两种：一种是直接指定的旅游网站，要搜索的网站列表应当存在某一类文件中；另一种是从大范围的网站中进行广义上的搜索，





对其中的网页内容进行过滤。其中第一类抓取内容比较专业，效率较高，应该重点关注，第二种作为辅助手段。需要提取结构化的信息。活动一般包括召集人，联系方式（例如 QQ 或手机号码），召集人数，集合地点和活动目的地等信息。

对于当前系统来说，爬虫系统的更新策略与所要抓取的网站优先级有关系，可以先对要抓取的网站进行优先级排序，对于优先级比较高的网站（比如专业的旅游网站）更新的频率应该高一些，而优先级较低的网站在系统资源紧张的情况下更新的频率可以低一些。

网页抓取的质量直接决定了网民对该网站的搜索体验，因此至关重要。主要有以下几方面组成：

A. 网页信息的过滤，爬虫在抓取网页信息的时候会住抓取下很多垃圾信息，对此必须进行过滤。关于过滤的方法暂时采取如下策略：对于可信度较高的网站，可以采取较为简单的方法进行过滤，而对于可信度较低的网站，就要采取复杂的方法或算法进行过滤以保证搜索的质量。

B. 网页去重，很多旅游网站上的信息是重复的，重复抓取不仅增加了系统的负担（需要消耗系统 CPU、硬盘等资源），更重要的是降低了用户对搜索结果的满意度，关于网页去重，有相关的对应方法和算法等，在此不详细描述。

爬虫运行日志记录了爬虫每天工作的过程，是改进爬虫抓取效率，定位各种异常问题的关键信息，所以日志信息至关重要。自动分析日志系统，进行数据挖掘，这个系统可以不间断运行也可以按指定时间段运行，以某一个时间段为单位，下载该时间段的日志进行分析，列出日志中的异常信息，并以合适的方式显示出来，便于管理员分析，排除问题，并制定出合理的抓取策略。

爬虫在抓取去网页信息的过程中，会遇到各种各样的问题，因此爬虫系统必须十分健壮，能够应付系统运行过程中出现的各种情况，必要的时候能够以损失部分功能为代价以保证爬虫系统的持续运行。

爬虫系统除了支持现有的数据格式和抓取协议以外，还必须能够及时支持新的数据格式和抓取协议等。这就要求系统在软件结构上有很好的可扩展性，实现功能上的组件化（既可以裁减，也可以增加）。每一个组件的功能是抓取某一类格式的数据并设置相应的开关，如果开关打开则该功能生效，如果关闭则此功能关闭。这样通过不断地增加组件实现爬虫抓取各种类型数据能力的不断增强。在实现的时候要用到工厂、抽象工厂等设计模式。

有必要在一定时间段内对系统资源占用情况进行监控（如网络带宽、CPU 占用率、内存占用率等），并以适当的形式展现出来。



## 10.2 信息提取

可以从网页信息中提取用户想要的相关信息，如相关联系人的 QQ、MSN、电话号码等相关信息。但是有效信息的格式千变万化，如包含 QQ 信息的文本“QQ: 450703138, 电话: 13581873858”这就需要使用信息提取的方式。

此次信息提取采用的是规则提取的方式，先建立相关字典，存储所要抓取对象的必要的特征信息（如前缀信息、后缀信息等），再根据所要提取的信息特征定义一些规则，用于提取用户所要定义的对象。

提取网页信息中 QQ 信息的步骤说明如下。

① 定义要提取的信息类型，代码如下所示：

```
public enum DocType {
    QQInfo,           //要提取的 QQ 号
    GuillemetStart,   //开始符号
    GuillemetEnd,     //结束符号
    Link,             //链接符号
    Num,              //数字
    QQPrefix,         //QQ 前缀
    QGSuffix,         //QQ 后缀
    English,          //英文字母
    Unknow,           //未知的
    Other             //其他
}
```

② 定义 QQ 的特征：QQ 号是一串 5 至 10 位的数字，在网页文本信息中，QQ 的前缀信息有可能是 QQ、q 我、qq 等，首先在硬盘上建立一个词典路径，如 D:/dic，在路径内建立若干文本文件，每个文本文件对应一个词类型。用于提取 QQ 信息，首先建立名为 QQPrefix 的文本文件，用于存放 QQ 号码的前缀信息。

③ 定义 QQ 前缀信息和 QQ 数字之间的连接信息和结尾信息，如冒号、空格、顿号等。我们分别建立 4 个文本文件：GuillemetStart 用于存放冒号、顿号等符号；GuillemetEnd



用于存放空格等符号；NUM 用于存放代表 QQ 的数字；Other 文件则用于存放不符合提取规则的信息。

④ 定义提取 9 位 QQ 号码的规则，代码如下所示：

```
lhs = new ArrayList<DocSpan>();
rhs = new ArrayList<DocType>();
rhs.add(DocType.QQPrefix);
rhs.add(DocType.Num);
rhs.add(DocType.Num);
rhs.add(DocType.Num);
rhs.add(DocType.Num);
rhs.add(DocType.Num);
rhs.add(DocType.Num);
rhs.add(DocType.Num);
rhs.add(DocType.Num);
rhs.add(DocType.Num);
lhs.add(new DocSpan(1, DocType.Other));
lhs.add(new DocSpan(9, DocType.qqinfo));
addProduct(rhs, lhs);
```

这是定义 9 位 QQ 信息的规则，其中 NUM 有 9 位，如果想定义抓取 5 位的 QQ 信息。则将 NUM 的数量定义为 5 个即可。

另外举一个提取网页信息中的召集人的例子列举如下。

① 先定义要提取的信息类型，代码如下所示：

```
public enum DocType {
    convenor,           //活动召集人
    CallerPrefix,       //召集人前缀
    CallerSuffix,       //召集人后缀
    Unknow,             //未知的
    Other               //其他
}
```

② 然后定义召集人的特征、前缀信息、连接信息和结尾信息。这部分和定义 QQ 相关的提取特征相似。

③ 最后定义提取召集人信息的规则，代码如下所示：

```
lhs = new ArrayList<DocSpan>();
```

```

rhs = new ArrayList<DocType>();
rhs.add(DocType.CallerPrefix);
rhs.add(DocType.GuillemetStart);
rhs.add(DocType.Unknow);
rhs.add(DocType.GuillemetStart);
lhs.add(new DocSpan(2, DocType.Other));
lhs.add(new DocSpan(1, DocType.convenor));
lhs.add(new DocSpan(1, DocType.Other));
addProduct(rhs, lhs);

```

实际运行时, 用户所要提取的信息是一次提取出来的, 所以以上提到的提取 QQ 号码和召集人相关的类型是定义在同一个枚举类型中的, 代码如下所示:

```

public enum DocType {
    QQinfo, // 要提取的 QQ 号
    Fixedlinetelephone, // 固定电话
    Mobiletelephone, // 移动电话
    Convenor, // 活动召集人
    PeopleNumber, // 活动人数
    EmailInfo, // E-mail 信息
    GuillemetStart, // 开始符号
    GuillemetEnd, // 结束符号
    Link, // 链接符号
    Num, // 数字
    Start, // 虚拟类型, 开始状态
    End, // 虚拟类型, 结束状态
    Date, // 日期
    QQPrefix, // QQ 前缀
    QQSuffix, // QQ 后缀
    MobilePhonePrefix, // 移动电话号码前缀
    MobilePhoneSuffix, // 移动电话号码后缀
    CallerPrefix, // 召集人前缀
    CallerSuffix, // 召集人后缀
    AttenderPrefix, // 参与人数前缀
    AttenderSuffix, // 参与人数后缀
    EmailPrefix, // E-mail 前缀
    EmailMiddle, // E-mail 中间字段
    EmailSuffix, // E-mail 后缀
    CityInfo, // 出发城市
    CityPrefix, // 出发城市前缀
    CitySuffix, // 出发城市后缀
    DetailAddressInfo, // 详细地址
    DetailAddressPrefix, // 详细地址前缀
    DetailAddressSuffix, // 详细地址后缀
    English, // 英文字母

```



```
        Unknow,                //未知的
        Other                  //其他
    }
```

这个枚举类型里定义了所要提取的 QQ、召集人、电话号码、城市、邮件和详细地址信息。只要分别定义相关的规则，系统在扫描一次文本后，会一次性地将上述要提取的信息提取出来。



## 10.3 活动分类

需要把抓取过来的户外活动分类。把户外活动分成：运动、徒步、自驾、游泳、登山、滑雪、骑行、休闲、聚会、旅行、户外、公益一共 12 个类别。

因为对一个活动的描述往往很短，所以采用简单的基于关键词匹配的分类方法。根据标题分类的实现代码如下所示：

```
String[] keywords = new String[] { "漫步", "夜游", "徒步", "步行", "拉练",
    "远足", "远行", "压路机" };
Rule r = new Rule(keywords, "徒步");
decisionList.add(r);
keywords = new String[] { "拼车", "自驾", "驾驶", "远征", "驾车", "开车", "
    驾", "搭车" };
r = new Rule(keywords, "自驾");
decisionList.add(r);
keywords = new String[] { "船潜", "潜水", "船潜", "船宿", "游泳", "潜泳",
    "游水", "仰泳", "蝶泳", "花样游泳", "蛙泳" };
r = new Rule(keywords, "游泳");
decisionList.add(r);
keywords = new String[] { "山", "登", "登山", "爬山", "登高", "高峰",
    "夜爬", "攀登", "登顶", "爬", "峰", "主峰", "雪山", "山峰", "上山", "岭" };
r = new Rule(keywords, "登山");
decisionList.add(r);
keywords = new String[] { "滑", "滑雪", "雪地", "滑冰", "溜冰" };
r = new Rule(keywords, "滑雪");
decisionList.add(r);
keywords = new String[] { "骑行", "骑车", "单骑", "单车", "自行车",
    "摩托车", "骑士" };
r = new Rule(keywords, "骑行");
decisionList.add(r);
```



## 10.4 搜索

根据行业特点设计出不同的搜索栏目。前期设计出自助游活动搜索、商业旅游线路搜索、旅游新闻搜索、旅行攻略搜索四大功能。采用迭代式开发方法，首先实现活动搜索，然后再实现剩下的三个，以后再考虑增加酒店搜索等功能。设计把不同的信息存放在不同的索引库中。自助游活动搜索、商业旅游线路搜索、旅游新闻搜索、旅行攻略搜索各自使用独立的索引库。

在活动搜索结果页面集成当地的天气信息。活动搜索结果页面显示发起人的联系信息，例如 QQ 号码，用户可以根据链接直接加 QQ。

在前期搜索访问量小，为了节省硬件成本，提高系统的运行效率，采用 Lucene 而没有采用 Solr。

1798 户外搜索引擎具有同义词检索功能，可以根据用户使用特点自定义大量同义词库，提高搜索全面性。如搜索“周一”，同样可以搜索到内容中有“星期一”的活动；如搜索“跑步”会有“夜跑”等信息；搜索“爬山”会返回“夜爬”等信息。

可以考虑增加一个手机浏览的 WAP 网站。数据来源一样，只是方便在较小的手机屏幕访问。



## 10.5 本章小结

本章介绍了一起走吧户外活动搜索 (<http://www.1798hw.com>)。

这个项目在最开始的时候，爬虫和搜索运行在同一台服务器，后来则分开成独立的爬虫服务器和搜索服务器。爬虫抓下来的数据形成索引后，把索引同步到搜索服务器。以后可以考虑采用 Solr，把前台界面和后台提供搜索数据独立出来。



## 参考资料

### 书籍

ERIK HATCHER 和 OTIS GOSPODNETIC. “Lucene in Action”

Bing Liu. “Web Data Mining”

Haralambos Marmanis and Dmitry Babenko. “Algorithms of the Intelligent Web”

Toby Segaran. “Programming Collective Intelligence”

W.Bruce Croft,Donald Metzler,Trevor Strohman.“Search Engines: Information Retrieval in Practice”

### 网址

网 址	说 明
<a href="http://lucene.apache.org/">http://lucene.apache.org/</a>	Lucene 主站点
<a href="http://nlp.stanford.edu/IR-book/">http://nlp.stanford.edu/IR-book/</a>	图书《Introduction to Information Retrieval》网络版
<a href="http://trec.nist.gov/">http://trec.nist.gov/</a>	是文本检索会议（TREC）的网站。TREC 为检索系统提供标准语料及标准方法，每年开一次评测会
<a href="http://www.sigir.org/">http://www.sigir.org/</a>	是美国计算机协会情报检索专业组（SIGIR）的网站。SIGIR 是一个学术界与工业界年年举行交流的盛会
<a href="http://www.nltk.org/book">http://www.nltk.org/book</a>	图书《Natural Language Processing with Python》网络版